FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

An Architecture for the Scalable Benchmarking of IoT Middleware and Streaming Platforms

Luís Zilhão

Mestrado Integrado em Engenharia Electrotécnica

Supervisor: Ricardo Morla

Second Supervisor: Ana Aguiar

July 23, 2018

An Architecture for the Scalable Benchmarking of IoT Middleware and Streaming Platforms

Luís Zilhão

Mestrado Integrado em Engenharia Electrotécnica

July 23, 2018

Resumo

Atualmente, o número de dispositivos ligados à Internet cresce todos os dias. Desde o mundo da domótica que procura a utilização de dispositivos inteligentes nas tarefas domésticas, até à monitorização de estados de saúdes de pacientes, cada vez mais se está perante uma realidade em que a maioria dos dispositivos estão ligados. Este é o mundo da *Internet of Things*, ou *IoT*. Com o seu aumento de popularidade, surge também a necessidade de gerir toda a informação que é produzida por este tipo de redes. Neste contexto, surgem os *middlewares*, implementações de software desenhadas para serem o intermediário entre os produtores de informação, e os consumidores. Existem várias soluções de *middleware* para este tipo de sistemas *IoT*, uma vez que os casos de uso são muito diversos. Isto leva a uma dificuldade de escolha da melhor solução de *middleware* para um determinado problema. Daqui surge a necessidade de efetuar algum tipo de avaliação objetiva e sistemática em diferentes casos de uso.

Nesta tese propomo-nos a criar uma arquitetura que permita a avaliação de desempenho, e a partir dela desenvolver uma plataforma para efetuar testes de desempenho. Os principais objetivos desta plataforma são facilitar a avaliação de desempenho de vários *middleware* de forma rápida e eficaz, e também fornecer uma base de comparação entre diferentes experiências. Procedemos a várias avaliações de várias soluções de forma a não só tirar algumas conclusões sobre cada *middleware*, mas também de maneira a validar a nossa plataforma e mostrar que tipo de informação pode ser extraída e quais os seus benefícios.

Abstract

These days, the number of Internet connected devices grows each day. From the world of domotics where the goal is to utilize smart-devices for domestic tasks, to the monitoring of patients' health status, we are coming ever closer to a reality where the majority of devices are connected. This is the world of the Internet of Things or IoT. With its increase in popularity, comes the need to manage all of the information that is produced by these types of networks. In this context, middlewares emerge, software implementations designed to be the middle-man between the data producers and consumers. There exist several different middleware solutions for these types of IoT systems, as their use-cases are very diverse. This leads to a challenge in selecting the best solution for a given problem. From here, rises the need to make some type of objective and systematic evaluation in different use-cases.

In this thesis we propose to create an architecture that allows performance evaluation, and from it develop a platform to make performance tests. The main goals of this platform are to ease the assessment of several middleware in a quick and efficient manner, and also to provide a basis for comparison across different experiments. We proceed to make several tests of different solutions so that we not only obtain some conclusions about each middleware, but also to validate our platform and show what type of information ca be obtained, and what their benefits are.

Agradecimentos

Quero agradecer aos meus orientadores, o professor Ricardo Morla e a professora Ana Aguiar, pela orientação durante todo o primeiro e segundo semestre, tanto na tese como nas cadeiras de redes, e ao Carlos Pereira pela ajuda em algumas questões pontuais.

Aos meus pais, pelo incansável apoio durante todos os meus seis anos de faculdade, para não falar dos doze de escola.

Aos meus irmãos, que acham que este curso é sobre torradeiras, agradecer os conselhos e as parvoíces todas.

Por fim, aos meus amigos, que estiveram sempre lá. Seja a jogar dardos ou a estudar para os exames na sala de redes. Seja a jogar Magic às três da manhã ou a jogar SpeedRunners até amanhecer. Ou simplesmente a tomar café. Sem eles, não havia paciência para fazer uma tese.

Luís

"It's the job that's never started as takes longest to finish."

Samwise Gamgee, in *The Lord of the Rings*

Contents

List of Figures

Abbreviations and Symbols

AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
CoAP	Constrained Application Protocol
CPU	Central Processing Unit
DSPS	Distributed Stream Processing Platforms
HTTP	Hypertext Transfer Protocol
I/O	Input/Output
IoT	Internet of Things
IP	Internet Protocol
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
M2M	Machine to Machine
MQTT	Message Queuing Telemetry Transport
QoS	Quality of Service
RAM	Random Access Memory
REST	Representational State Transfer
RTT	Round Trip Time
SSL	Secure Sockets Layer
TCP	Transmission Control Protocol
TLS	Transport Layer Security
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
VM	Virtual Machine
XML	Extensible Markup Language

Chapter 1

Introduction

1.1 Context

The internet of things is an evolving world of networked devices, ranging from small sensors to home appliances with the goal of sharing data across several platforms and applications, from health care, by providing real-time monitoring of patients' vital signs, to home security systems, in order to monitor and control security by means of another networked device, such as a computer or phone [1]. Its origins can be traced back to 1990 when John Romkey connected a toaster to the Internet, allowing it to be turned on and off remotely. In 1999, the term "Internet of Things" was coined by Kevin Ashton of Procter & Gamble [2]. Since then the commercialization of IoT has increased immensely, with estimates of 26 billion devices in use by 2020 and trillions of dollars in generated revenue [1].

An IoT system is made up of several components, which may vary in accordance with the type of application it is being used for. Sensors are usually responsible for the generation of data, and these may vary in quality and also in structure of the data that is being generated, as there are a great number of device types, data types and potential providers in the IoT universe, which leads to a challenge in order to ensure interoperability across different platforms [3]. To achieve this, common frameworks and standards are required to ease development of new solutions. However, there is a lack of common standards that is hindering their development. A common ground must exist between different systems when it comes to protocols, message formats and sequences, and this is where middlewares come in.

1.2 Motivation

Middlewares in IoT aim to bridge the game between the data producers and the data consumers. With the rise of the Internet of Things, comes the need for different applications, and different services with different requirements. Each of these applications will ideally want its own middleware and sensor network so that they can be suited perfectly to their needs. In practical terms this is not possible, as it would be a waste of resources. Therefore, applications will need to work with

existing sensor networks and will either develop its own middleware, or choose an existing one that better suits their needs. The question then becomes: how to choose the best one for the task at hand? There are a great number [4] of available middlewares to choose from, which makes the selection process very time-consuming and impossible to be exhaustive. A comparison must be made between them to evaluate which is better suited for which task. But then comes the problem of how to make the evaluation, as performance measuring is not trivial, and common ground must exist for the comparison to be valid. Furthermore, since we have a great number of them, ensuring such common ground will not be possible across different experiments, and different researchers. From these difficulties arises the need for such common ground, a platform that enables multiple comparisons across different middlewares in an efficient manner.

1.3 Contributions

In this thesis, we set out to answer a question that is raised by the increasing number of middlewares available, which is: can we create a more streamlined process for middleware benchmarking?

In our quest to answer this question, we have created an architecture and subsequent framework that aim to ease the process of middleware benchmarking. Our work is, then, an aid in the benchmarking process specifically regarding publish/subscribe middlewares, with a focus on IoT. The selection process that drives the necessity for benchmarking has proven to be very time-consuming. Therefore, our platform will help future researchers to determine the behavior of these solutions, and more easily reach conclusions about which middleware is best for which use-case.

In the early stages of our development, we also wrote a paper that highlighted the strengths of our approach, as well as identified some core weaknesses that we have addressed in this thesis [5].

1.4 Thesis Structure

After this introduction, in the second chapter, we will start by presenting some background knowledge on middlewares and the publish/subscribe communication model, as well some related work focusing on benchmarking and specifically middleware benchmarking.

In the third chapter we will show the main problem that we need to address. We will start by showing the difficulties in middleware selection, followed by our proposed solution and its requirements. Lastly, we will justify our selection of middleware, as well as providing a brief explanation regarding each of their structures and how they work.

In the fourth chapter, we will present our solution in detail. It is divided into two main sections, each corresponding to an iteration of our solution. In the first section, we will provide insight into the development process as well as the resulting structure and its limitations. The following section will address the previously defined limitations, and present a new structure, along with its own limitations.

The fifth chapter will be comprised of the results that we were able to obtain through several benchmarking processes. There will be several different comparisons, across different middleware and protocols, each with its own discussion and information regarding the benchmarked middlewares, as well as our platform itself.

Lastly, in the sixth chapter we have the conclusions that we were able to draw from the development process, as well as what we were able to achieve with our framework and its limitations. We also present a brief analysis on the future work needed to address these limitations.

Chapter 2

State of the Art

2.1 Background

2.1.1 Publish-Subscribe Communication Model

Publish-subscribe [6] is a messaging pattern were the subscribers express interest in certain events and are notified when they are generated. These events may by grouped by certain categories which enable the subscriber to be notified only when events occur of a category of their interest. The way that this works is by introducing a broker between the data producers, which are the publishers, and the data consumers, which are the subscribers. This allows the subscribers to tell the broker what data they are interested in, and when the publisher sends such data, it will notify only the subscribers which are interested.

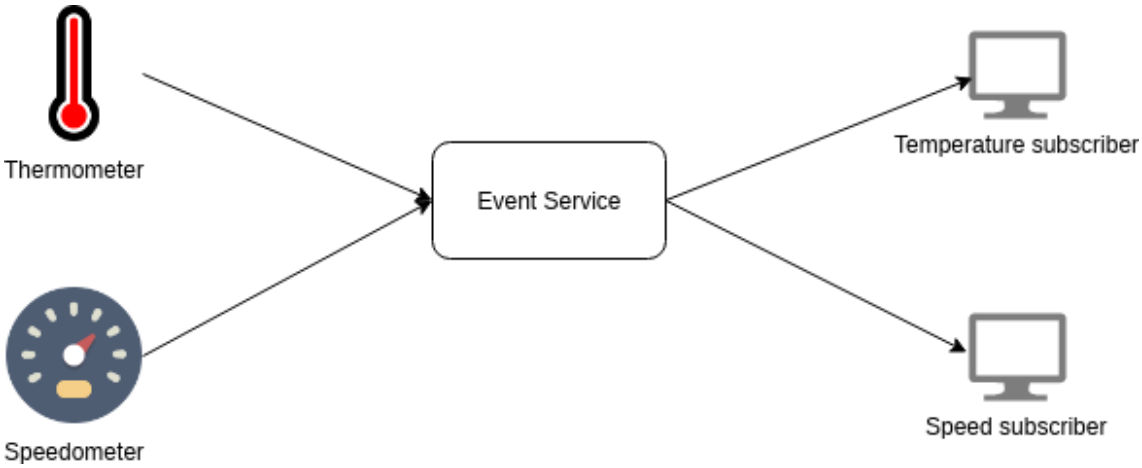

Figure 2.1: Publish-Subscribe model

Let's take the example in figure 2.1, a topic based publish-subscribe scheme. We have a certain service that is interested in using temperature information of a certain location and another which is interested in the speed of a certain car. Since these are totally different purposes and each service is not interested in any other information besides their specific metric, it would be a waste of resources to send information which is not required. The event service will then ensure

that only the temperature subscriber receives temperature updates, and the speed subscriber will receive speed updates. This differs from the request-response model typically used by HTTP as the publishes are not sent to specific receivers, but to an intermediary who will then notify the interested parties of any event. Therefore, the publishers have no knowledge of who is receiving the messages, and similarly, the subscribers have no knowledge who is publishing the information. This provides scalability and a dynamic network topology.

2.1.2 Middlewares

Middlewares are hard to define precisely and can be used across a variety of different industries and can be grouped into several different categories. We can define middleware as being a general-purpose service that lie between platforms and applications [7][8]. Here, we are interested in communications, specifically message queuing. In this case, the platforms consist of low level services and processing elements [7], being the "things" that generate the data, and the applications are the services that will use that data, and transform it into knowledge that is fit for human consumption. There are several challenges mentioned in [8] that middlewares aim to resolve in

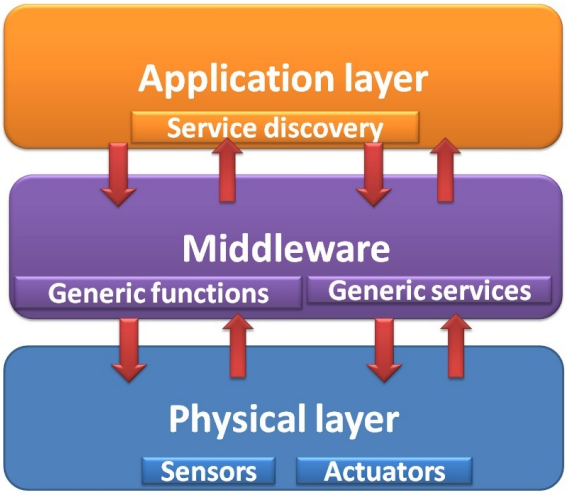

Figure 2.2: Middleware Structure [8]

IoT, and for the purpose of this thesis we will focus on a few:

- **Interoperability** — As it has been mentioned, middlewares must provide a way for different devices to interact and collaborate.
- **Scalability** — They must be able to efficiently handle an increasingly large number of devices.
- **Heterogeneity abstraction** — A higher level of abstraction is required to hide the low-level communication complexity.
- **Security** — Data security is of the utmost importance if we consider IoT systems that will control critical applications, to ensure privacy and integrity.

- **Extensibility** — Due to the ever evolving nature of IoT, the possibility to integrate new technologies must be considered.

2.2 Related Work

2.2.1 The Benchmarking Process

The best choice of benchmarks to measure performance is real applications [9]. Indeed the best way to measure the performance of a certain platform and to ensure that it meets certain criteria is with the actual applications that will be using said platform in the same conditions. However it is not always possible to have a complete setup available for testing purposes, and this brings about the need for platforms whose sole purpose is to evaluate performance, by attempting to emulate real world scenarios and workloads and measuring their performance. But first, we need to define what performance is. That means establishing metrics that are relevant to the platforms which are being tested and that are able to convey useful information to the user and best predict the behavior in real world situations.

Let's take the example in [9] were they are attempting to measure computer performance. Here, we are interested in defining what it means for a computer to be faster. The user is interested in response time, so they then define performance as being the inverse of execution time:

$$\text{Performance}_X = \frac{1}{\text{Execution time}_X} \tag{2.1}$$

And create a relation between executions times of two different machines, X and Y:

$$n = \frac{\text{Execution time}_X}{\text{Execution time}_Y} \tag{2.2}$$

Of course, this definition of time may not be the most useful, as it measures elapsed time and does not take into account the time while waiting for I/O, which leads to the creation of another measure of time, CPU time. Even with this simple example, we can see that the same concept can have different meanings, as the user is interested in elapsed time, however with multiprogramming we are interested in CPU time. After defining metrics, comes the need to define a workload that mimics real world usage, and under which conditions it will run. The authors raise the question of allowing source code modifications that would improve performance, and that these should only be allowed if they translate to real world benefits, which leads to the creation of benchmark suites, that use a variety of applications and metrics to complement each other. The benchmarks are divided between desktop and server, as they have different use cases and require different workloads. When it comes to results, one important concept also mentioned is reproducibility, which means detailed descriptions of everything required to be able to replicate the results. When presenting the findings, it is also useful to summarize the values obtained, so as to not overload the user with too many separate values, and instead attempt to provide a single metric which is

composed of several others in the benchmark suite, and they achieve this by using a geometric mean of the same metric across different workloads.

Sangroya et al. [10] present an architecture for dependability and performance benchmarking of cloud services. They argue that guaranteeing reliability, performance, and availability are major challenges for cloud services providers, and that there exists no framework to properly evaluate these aspects. Some of the key aspects they mention as being necessary for a dependability benchmark are: representativeness, repeatability, portability, and scalability. As for their framework, they divide it into four phases. First, the workload is specified by the user. In the second phase, the load is injected by the user into the system. The third phase is the monitoring phase, where all the statistics are calculated and stored. In the fourth and final phase, the statistics are processed to produce graphs for better readability.

2.2.2 Middleware Benchmarking

In [11] the goal was to obtain a set of qualitative and quantitative metrics that are suited for IoT middleware benchmarking and develop a test methodology around those metrics. Following this, the authors were able to use said methodology by making a comparison of two IoT middleware platforms, FIWARE[1] and ETSI M2M[2]. They used a smart cities scenario which is fairly common in IoT which typically uses the publish-subscribe communication model. The platforms themselves were treated as black-boxes, disregarding information about internal implementation. This eases not only the process, but also the creation of a common benchmarking platform. This can cause the middleware to be used sub-optimally, but is a necessary step in an effort to generalize the process, and should also be the most common situation in real world usage. A qualitative analysis was conducted with the goal of identifying certain middleware functionalities which are relevant for IoT applications and ease their development, such as documentation availability and which communication models are supported. This type of analysis and metrics requires a case-by-case look on each of the platforms being compared, and while they cannot be implemented as part of a benchmarking platform, as they are not measurable automatically, they can still take part in the global architecture. A quantitative analysis was performed using specific metrics relevant for communications scenarios, particularly mass publication of resources using a publish-subscribe model, such as publish time, which is defined as the elapsed time since sending the HTTP request and receiving the HTTP response. Of course, the measurements are specific to this type of communication model and protocol, which we are trying to avoid in this thesis, by providing general metrics that are protocol independent. For protocols that share a communication model, such as request-response, this should be relatively straightforward, but it could be more complicated in other cases, for instance with HTTP and MQTT.

In [12], the authors attempted to recreate and improve the previous experiment using a controlled environment, but using OM2M, which is an implementation of the oneM2M[3] standards. A

[1] https://www.fiware.org/
[2] http://www.etsi.org/
[3] http://www.onem2m.org/

java application was created for each of the four setups they aimed to measure: FIWARE, OM2M HTTP, OM2M CoAP, OM2M MQTT. They raise a few concerns which should be had when attempting the benchmarking process, such as where and how the data is published, which mainly will concern application development. This is a point which we want to address in this thesis by developing a module for a certain middleware and re-utilizing it for subsequent measurements. Another question raised is the enabling of multi-connection capable clients, which might create an uneven playing field if different clients are used. With the creation of a unified platform, this will have to be taken into account for the client development. Finally the Java configurations, which address the issue of different optimizations with different default Java Virtual Machine packages, which might vary across operating system and also depending on if they are client or server VMs.

In [13], a benchmark suite is developed for Distributed Stream Processing Platforms for IoT applications, along with relevant metrics. The tasks that are usually performed in these types of applications were classified, and based on the classification they classified the applications themselves:

- Extract-transform-load and archive operations
- Summarization and visualization
- Prediction and pattern detection
- Classification and Notification

They then proceeded to identify the metrics which are most relevant for evaluating performance on DSPS:

- Latency
- Throughput
- Jitter
- CPU and Memory utilization

These are relatively common metrics and can be used across different communication scenarios, be they request-response or publish-subscribe, and can therefore be used in this thesis alongside others.

Following this, they used the tasks to define micro-benchmarks to evaluate performance on individual tasks, with the most relevant category for this thesis being the IO tasks, as these are the type of tasks performed by message queuing systems which our platform aims to evaluate. Even though not all categories are relevant, we can adopt the same methodology and define classes for our workloads, with the aim of evaluating different metrics and simulating different communication scenarios. The workloads themselves are four data streams available to the public. Each of them has a different number of sensors, message size, distribution, and rates. From these data sets,

they use scaling factors to increase or reduce the message rates and also the number of sensors, providing flexibility in terms of possible workloads. We can adopt this in our platform by having a few parameters which are user-controllable, thus enabling the dynamic creation of workloads that better mimic the behavior of a desired IoT setup.

In [14], the authors analyze which are the main requirements of IoT platforms, and which pub/sub solutions provide support to these features. Following this, they perform an evaluation with four popular pub/sub solutions: rabbitMQ, mosquitto, ejabberd, and ZeroMQ. They define three main types of decoupling, especially suitable for large-scale IoT deployments:

1. Message producers (publishers) and consumers (subscribers) need not be connected at the same time

2. Messages are not addressed to a specific consumer, but rather to an symbolic address, such as a topic

3. Messaging is asynchronous, non-blocking

The authors mention that there are many pub/sub solutions for cloud IoT settings, yet little reliable data exists for which one will be the better fit for IoT requirements. They go on to establish three reference scenarios:

- Social Weather Service: Sensor device owners share their sensors with the public

- Smart car sharing: Cars can periodically offer themselves for rental and provide additional information

- Traffic monitoring: Cameras take pictures of cars and send them to the cloud

After establishing these reference scenarios, the authors go on to outline the IoT requirements that mus go into account when undergoing the middleware selection process. Those requirements are:

- Messaging Pattern: One-to-one and one-to-many messaging patterns should be supported

- Filtering: Subscribers should only receive what they are interested in

- QoS Semantics: The middleware should be able to classify subscriptions and messages with a QoS level

- Topology: Middlewares must support a centralized topology

- Message Format: Middlewares must be payload agnostic

2.2.3 Analysis of current solutions

The previously shown sources such as [11] and [12] have benchmarked several pub/sub platforms, and established metrics on which they should be compared. In order to benchmark them, a small application was developed for each platform. The problem with this approach is that it is not easily scalable as we increase the number of middlewares. In [14], the authors also performed a qualitative and quantitative evaluation on a few middleware platforms, and highlighted the main requirements of IoT platforms.

The problem with the previous benchmarks lie mainly with the scalability and comparability aspects. While they are very useful to extract information on the chosen middlewares, if someone needs to make a new comparison with an additional platform, they will have to implement it from scratch, with no re-utilization of features between different middlewares. Also, between the different researchers, different approaches were used, such as metrics and types of load, which means that a comparison cannot be made across different studies. The main problem, then, is that no scalable benchmarking platform exists to solve these issues, which is what we aim to develop in this thesis.

Chapter 3

Problem and Methodology

3.1 Characterization

As we have seen, middlewares for IoT exist in great number [4] and each of them have their own specificities when it comes to implementation. With this, comes a great difficulty in performing exhaustive tests, as it is not feasible to create an application for each of them. Also, even if some are created for benchmarking attempts, the results may not be comparable across experiments if the conditions are not the same, something which is likely to happen if they are done by different users. Next comes the question of which types of message patterns are supported, such as request-response or publish-subscribe. Each of these represent a different way of interaction with the middleware, which implies the message exchange is also different. In addition to the message pattern, there are also the different protocols that each middleware supports. Let's take for example FIWARE and OM2M. Both of them support HTTP as a communication protocol, enabling a more direct comparison. However, OM2M also supports CoAP and MQTT. In the case of HTTP and CoAP, both are request-response [15, 16], and therefore share a common message pattern, where the client always contacts the server first, and then waits for a reply. However, this is not so clear once we consider MQTT, which is a publish-subscribe messaging protocol[1].

Let's consider a publish-subscribe system such as figure 3.1. It is an intrinsically different situation, as the publishers do not communicate directly with the subscribers, using a middle-man to manage communication. How can we compare this in a fair manner to a request-response pattern with the same use-case? Another problem is raised when we attempt to implement a publish-subscribe scenario using a request-response protocol such as HTTP, as we now have to compare the same protocol, over different messaging patterns.

3.2 Proposed Solution

To solve these issues, a common setup must exist to, not only ensure the same conditions across several benchmarks, but also ease the process by providing modularity across the components and

[1] http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/ mqtt-v3.1.1-os.html

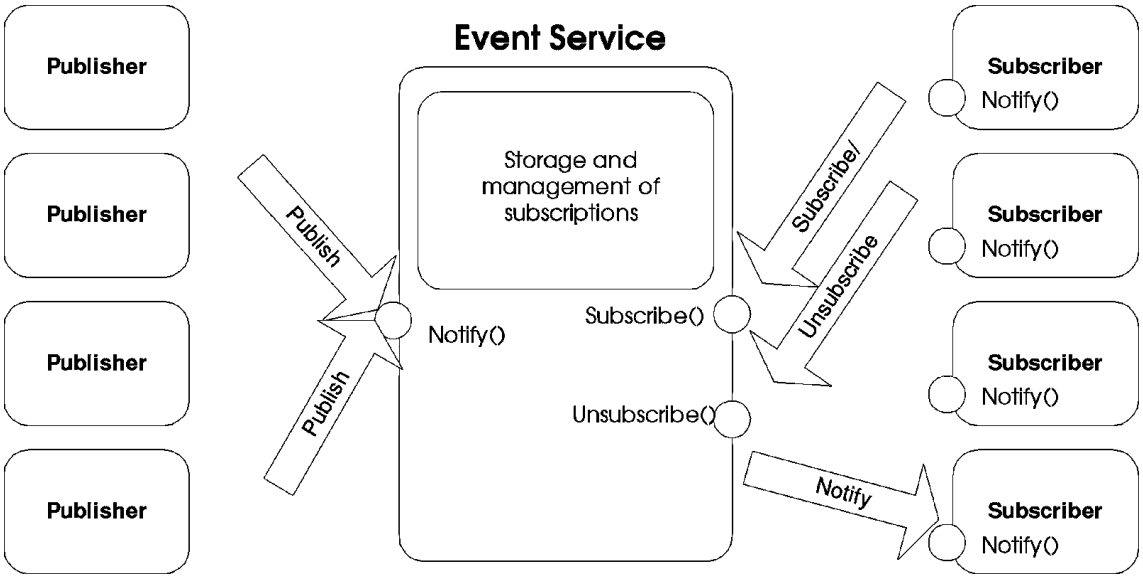

Figure 3.1: A typical publish-subscribe system [6]

enabling scalability. What we propose, then, is a modular architecture that attempts to generalize the process, and factorize where possible each of the different steps. From this architecture will stem a platform that aims to provide benchmarking tools for a few existing middlewares using certain protocols and messaging patterns. However, the main feature of this platform, will be its capacity to incorporate new middlewares and protocols as needed. This will permit a user with a new middleware to easily implement the necessary requirements for their new solution into the platform and re-utilize its existing resources, therefore, not only speeding up the process, but also ensuring the same conditions across all middlewares.

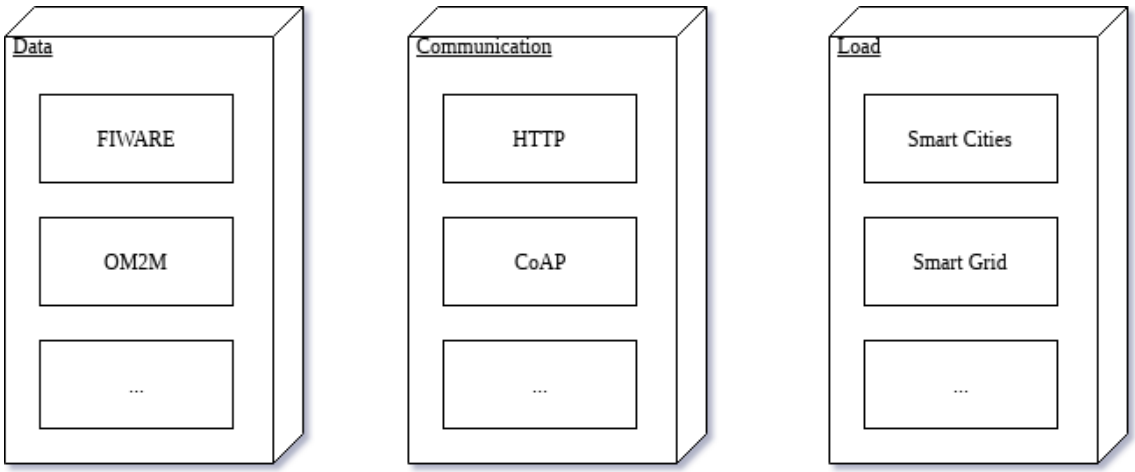

Figure 3.2: Main blocks in our architecture

The main building blocks that we propose are present in figure 3.2. The goal is to only implement what is necessary for a new entry in the benchmarking platform instead of using a similar implementation for each one. We propose a communication block where the communication pro-

tocols are lodged, such as HTTP or CoAP, and each has the methods implemented, such as POST or GET, so that they are totally platform independent and can be reused. If a new protocol is required to be added to the platform, its methods can be implemented without interfering with the remaining structure.

The data block is where the middleware specific functions reside, and each of these is responsible for implementing its data structure and bridging the gap to the protocols. Similarly to the data block, it is designed so that each is independent so that all can use the same communication methods implemented in the communication block. With a new entry, one can observe how the existing blocks are structured, thereby speeding up the process and keeping it isolated without interfering with the existing setup.

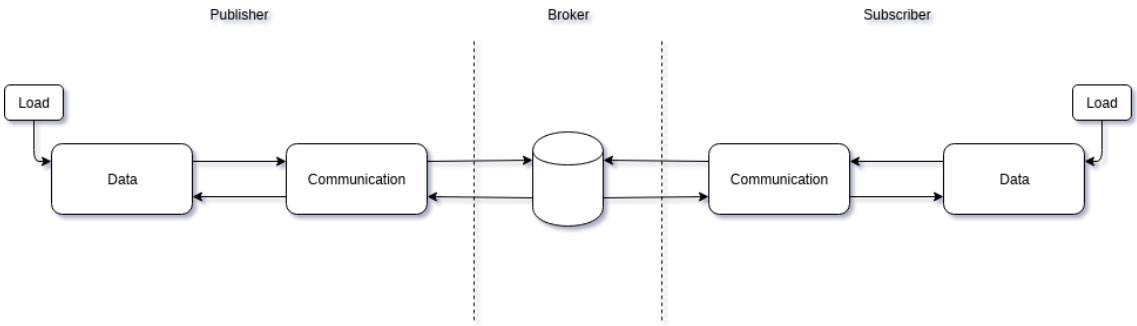

Figure 3.3: Basic building block structure

The load block will enable different types of IoT scenarios to be programmed and dynamically changed, so that we can attempt to mimic real world scenarios such as Smart Cities. Again, this should be independent from each of the other blocks so that the same workloads can be used throughout all middlewares and protocols, providing a basis for comparison and ensuring high flexibility.

3.2.1 Requirements

In order for out platform to be usable, a certain set of requirements will have to be met. They refer to the two main use-case scenarios for our platform, which are: addition of a middleware to the platform, and the benchmarking of existing and already supported middlewares.

When a developer comes into contact with our platform, the first immediate question that they will ask is: what do I need to do to add a middleware to this framework? We must then address this question, by providing the instructions necessary for a new addition. Referring again to figure 3.3, if a new middleware is presented, the user will need to implement the data block, as these will correspond to the methods which assemble the necessary structures for publishing and subscribing to resources. These are not possible to factorize as each middleware has a different implementation and specification. However, when implementing this block, a user should not be worried on affecting the rest of the system. Therefore, each block must be independent from each

other, and must have generic inputs and outputs. The inputs should refer to properties such as message size, and the outputs should be structures to be used as payloads for the next block.

Following the development of the data block, come the communication protocols. These should be generic clients so that they can be used across all middlewares. When the protocol has already been implemented, a user should only need to implement the link between the data block and the protocol. If we take HTTP as an example, this link could mean specifying the headers that should go in each request. Since this is not information that is part of the middleware data structures, and has intrinsic connections to a given protocol, it cannot belong to the data block. On the other hand, because it is also intrinsically tied to a certain middleware, it cannot be considered a generic component, thus it cannot belong to the communication block.

The load block is where we define parameters such as message delivery rate and message size. These should be easily configurable by the user by way of an argument or a variable. Each load scenario corresponds to a certain distribution of message delivery. The first and default scenario is a linear one, where messages are sent at a constant rate defined by the user. If the user wishes to send messages with a different distribution, they should create a new scenario under the load class. For each scenario to be usable across any middleware, it should only control when a publish should occur.

For the metrics, these must again be generic and not depend on any specific middleware detail. Hence, whichever metrics we decide to implement must only require data that any middleware implementation can provide. As long as the previous blocks have been developed with the appropriate inputs and outputs, the user should not need to develop anything for the metrics, as these are calculated with the generic data, provided earlier. On the other hand, if the user decides to add to the existing metrics, this may have implications of the data that must be registered by the previous blocks. This means that in order to add metrics that rely on data which is not being registered at a given time, it may be required to change output values along several blocks.

3.3 Middleware Selection

For this thesis, the middlewares that we have selected are: FIWARE, OM2M, Ponte and RabbitMQ.

- FIWARE was chosen as it had been used in previous experiments we and already had a good degree of familiarity with it. In addition to this, it implements the required publish/subscribe model, and is an IoT platform which can be considered state-of-the-art [17].

- OM2M was selected mainly due to being a reference implementation of the oneM2M standard. With this being a specification that developers are aiming to integrate into IoT solutions [18], we considered it to be a good choice to develop our framework. Similarly to FIWARE, previous work had been conducted with OM2M, so some familiarity with it was already present, which further increased the motivation for us to use it.

- After the implementation of FIWARE and OM2M, three communication protocols were implemented in the framework: HTTP, CoAP, and MQTT. With this being the case, we considered Ponte to be a good addition to our framework as it would allow a comparison to be made with the remaining middlewares with all three protocols.
- RabbitMQ is a popular implementation [14], and has seen use in IoT frameworks and implementations [19]. Due to its popularity as a message broker, we felt it would be a good addition.

In remainder of this section, we will provide a brief explanation of how each middleware is structured, and what types of communication protocols it supports.

3.3.1 FIWARE

The FIWARE middleware has a series of components, one of which is for data management called Orion Context Broker[2]. It uses a publish-subscribe messaging pattern over HTTP. Data is sent in using a JSON structure, such as in listing 3.1.

```
{
  "id": "Room1",
  "type": "Room",
  "temperature": {
    "value": 23,
    "type": "Float"
  },
  "pressure": {
    "value": 720,
    "type": "Integer"
  }
}
```

Listing 3.1: JSON payload for entity creation

Orion uses a several context management operations, we will focus on a few which are the most basic and relevant for our setup:

- Entity creation
- Update entity
- Subscription

Previously we saw the basis of the publish-subscribe communication model, and how it categorizes messages so that each subscriber can choose what they are interested in receiving. Here, they call those categories entities. Entity creation is done thorugh an HTTP POST request. Taking again the example in listing 3.1, we can see the creation of an entity with id “Room1”, and a few attributes such as “temperature” and “pressure”. Now say we wanted to subscribe to this entity, we can see an example of this in 3.2.

[2]https://fiware-orion.readthedocs.io/en/master/user/walkthrough_apiv2/index.html

```
{
  "description": "A subscription to get info about Room1",
  "subject": {
    "entities": [
      {
        "id": "Room1",
        "type": "Room"
      }
    ],
    "condition": {
      "attrs": [
        "pressure"
      ]
    }
  }
}
```

Listing 3.2: JSON payload for a subscription

The subscriber need to identify not only the entity it wants to receive information about, but also which of its attributes. Note that it will only receive notifications upon a change in the values. Also, since the value is simply overwritten, the broker is memoryless. But how are these updates generated? It can be done through the HTTP PATCH operation with the payload visible in listing 3.3.

```
{
  "temperature": {
    "value": 26.5,
    "type": "Float"
  },
  "pressure": {
    "value": 763,
    "type": "Float"
  }
}
```

Listing 3.3: JSON payload for a update

The entity is identified through the URL in the patch request. An update to a single attribute can also be achieved through a PUT request with the values in the URL.

3.3.2 OM2M

OM2M[3] is an implementation of the oneM2M standards and, like FIWARE, implements a publish-subscribe model. Data can be sent using HTTP, CoAP or MQTT, however, for the latter an external MQTT broker, such as mosquitto[4], is required. Data can be structured using XML or JSON.

As we have seen previously in the publish-subscribe model, information is categorized and subscribers only receive information pertaining to what they want. In FIWARE, we saw that those categories are called entities, whereas here they are called applications. To create an application, the message structure can be seen in 3.4.

[3] http://www.eclipse.org/om2m/
[4] https://mosquitto.org/

```
<?xml version="1.0" encoding="UTF-8"?>
<m2m:ae xmlns:m2m="http://www.onem2m.org/xml/protocols" rn="MY_SENSOR">
    <ty>2</ty>
    <ri>ae-CAE449907766</ri>
    <pi>/in-cse</pi>
    <ct>20151104T151858</ct>
    <lt>20151104T151858</lt>
    <lbl>Type/sensor Category/temperature Location/home</lbl>
    <acpi>/in-cse/acp-89704715</acpi>
    <api>app-sensor</api>
    <aei>CAE449907766</aei>
    <rr>false</rr>
</m2m:ae>
```

Listing 3.4: XML payload for application creation

Next, we have another concept which are the containers. These do not have a strict parallel in FIWARE, but we can think of them as being the attributes, as we can create different containers for different data types, such as temperature or pressure. These are created once. The publishes themselves occur by creating what they call content instances, an example of which can be seen in listing 3.5.

```
<m2m:cin xmlns:m2m="http://www.onem2m.org/xml/protocols">
    <cnf>message</cnf>
    <con>
      &lt;obj&gt;
        &lt;str name=&quot;appId&quot; val=&quot;MY_SENSOR&quot;/&gt;
        &lt;str name=&quot;category&quot; val=&quot;temperature &quot;/&gt;
        &lt;int name=&quot;data&quot; val=&quot;27&quot;/&gt;
        &lt;int name=&quot;unit&quot; val=&quot;celsius&quot;/&gt;
      &lt;/obj&gt;
    </con>
</m2m:cin>
```

Listing 3.5: XML payload for contentInstance creation

They are stored in succession, not overwritten, which means this middleware stores previous states in memory, a noticeable difference from FIWARE.

3.3.3 Ponte

Ponte is a multi-transport Internet of Things / Machine to Machine broker [20]. It supports MQTT and REST APIs over HTTP and CoAP. It allows for inter-protocol communication, such as publishing with MQTT and notifications through HTTP. It doesn't have any specific structure to send, as all the information regarding the target resources or attributes go in either the URL or the topic. This makes it relatively easy to implement. The usage of this middleware is as follows:

- **MQTT**: To publish using MQTT we merely specify the topic where we want to publish and send the message as payload. In our case, the topic will be of the form resourceX/attributeY, X being the resource number and Y being the attribute number. The message to be published need only be the sequence number, as Ponte requires no other information nor message

format such as a JSON. To subscribe to a resource, we only need to make a subscribe request to the given topic.

- **CoAP**: For a publish request with CoAP, a PUT request is made to the target resource and attribute. These are specified in the URL, so the payload needs no other information other than the message itself. For a subscription, we will need to make an observe request for the target URL.

- **HTTP**: Publishes behave similarly to CoAP and are made with PUT requests. However, there is a limitation when it comes to the subscriptions, as Ponte with HTTP only allows us to perform successive GET requests to know the status of our resources. This means it deviates from the publish/subscribe method, and subscribe time will be limited by the polling rate.

3.3.4 RabbitMQ

RabbitMQ is the most widely deployed open source message broker[5]. Along with its popularity, we saw a reference to how it could be used for IoT scenarios in the related work section, which makes this a good candidate to add to our platform. It is a very versatile broker with its core protocol being AMQP, and also supports other protocols such as CoAP and MQTT through plugins. HTTP is only supported as a management protocol.

In terms of the application itself, RabbitMQ offers libraries for a developer to build their own client. This makes developing a client easier, but makes it harder to use a generic client for communication, as it is not as clear how to use the API. Here we are presented the two options: to attempt to take advantage of the existing clients already implemented in the framework and attempt to implement the RabbitMQ structure, or to use the framework as a wrapper for the provided libraries. The first approach is more desirable as it puts RabbitMQ on the same level as the previous middlewares, and gives us total control over the structures and measurements. However, it is more time-consuming as we need to understand how each protocol API works and implement it. While the second approach may be faster, we will lose on some of the measurements such as the publish structure size. This is due to the fact that the provided publish methods don't separate the creation from the sending of a publish request. Therefore, we have no access to the structure and cannot analyze it.

Its structure is the AMQP model which we can see in figure 3.4. Messages are published to exchanges, and then routed to queues. The messages can then be delivered to subscribers of those queues. This means that to implement RabbitMQ in our framework, we need to map the exchanges and queues to resources and attributes, in order for it to have a similar structure to the other already implemented middlewares.

[5]https://www.rabbitmq.com/
[6]https://www.rabbitmq.com/tutorials/amqp-concepts.html

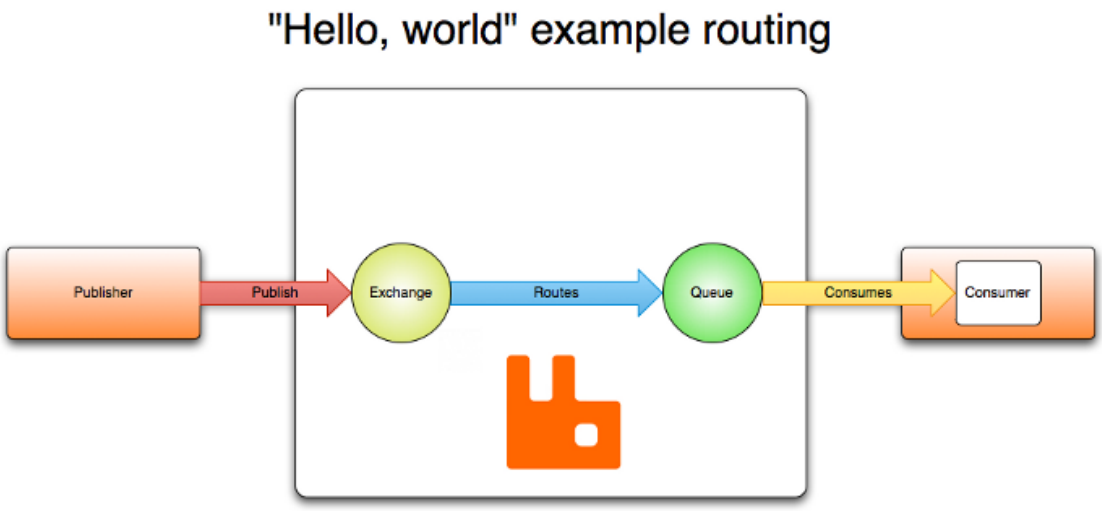

Figure 3.4: AMQP routing example[6]

Chapter 4

Solution

The proposed architecture and the platform that it stemmed, did not arise from a single development cycle. Rather, it was an iterative process where we crafted an initial solution, documented the weaknesses, and identified the areas which could be further factorized. In this chapter we will present the iterations that led to the current state of the platform, and what changes each one introduced.

4.1 First iteration — The First Draft

The existing code on which the platform is based on comes from a previous experiment conducting a comparison [12] between FIWARE and OM2M and had been implemented in Java. However, it was very middleware and protocol specific, with no easy way to separate the two and also no way to add a new one without starting almost from scratch. To avoid this, we needed to generalize the existing code, and create some structure so that future additions could re-use existing code, and also have guidelines that could ease the process.

4.1.1 Structure

The generalization of the code has so far resulted in the classes visible in figure 4.1. A superclass Middleware implements three generic methods which should be implemented by all middlewares, regardless of their structure, and any attributes that are common as well, such as the URI. The first two methods are **publish()** and **subscribe()**, which are self-evident in publish/subscribe scenario. Then, we have **destroy()**, whose goal is to delete all the existing resources of a broker. This will allow the broker to start the resources from scratch and it will make it easier to conduct successive experiments be deleting everything at the start. The **listen()** method creates a listener for the notification requests, and registers the times at which they arrive, so that we may calculate the subscribe time. Finally, **createInitialSetup()** creates the desired number of resources and attributes in the broker, so that they may be used for publishing.

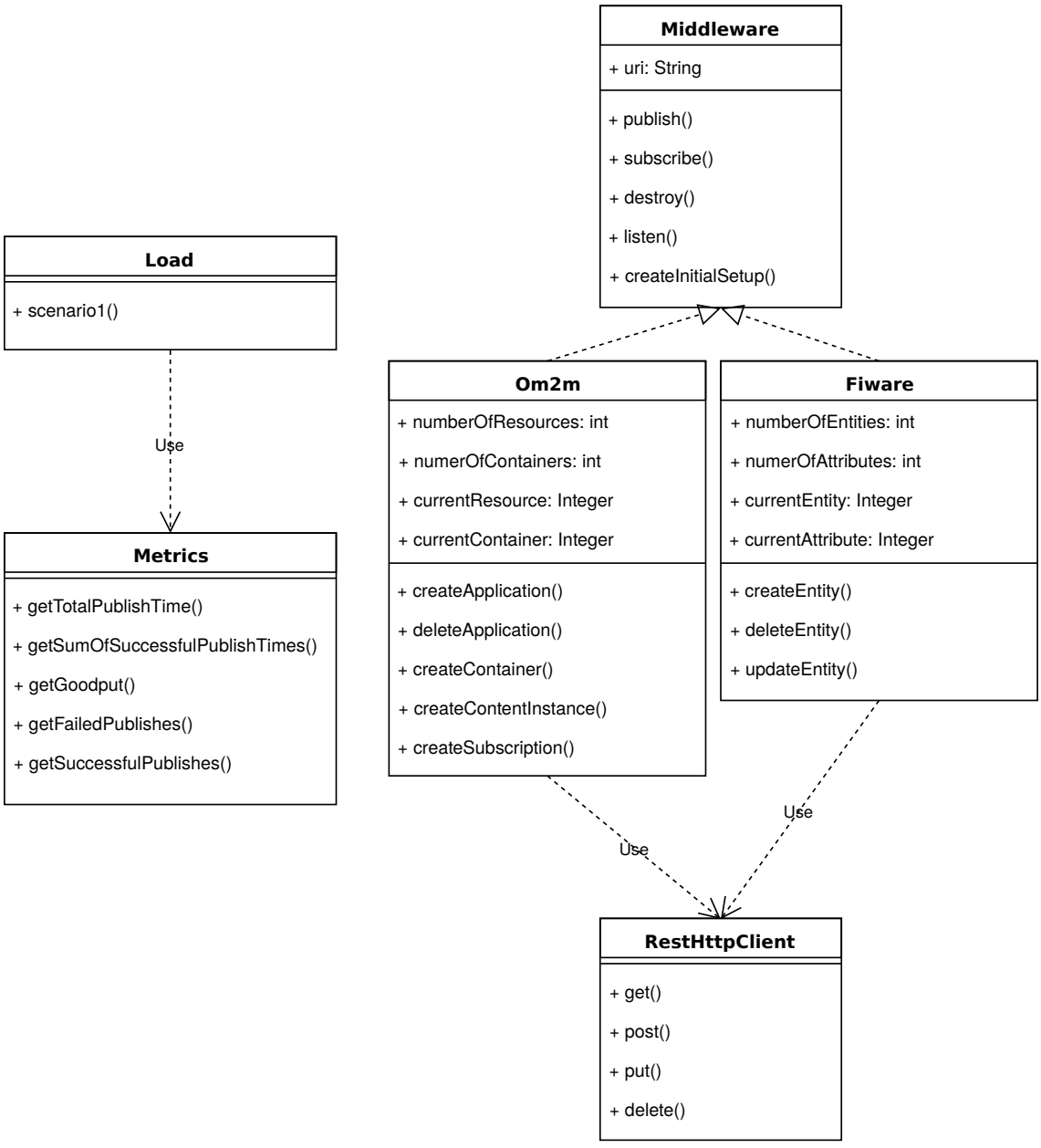

Figure 4.1: Class diagram for the first iteration

The load class is responsible for defining the attributes of our load, such as the size of messages and how many they will be. It will register any information pertaining to the requests and pass it down to the metrics class, which will then calculate the implemented measures for our framework.

Then, two middleware classes currently exist, OM2M and FIWARE. Each of these will extend the previously defined middleware superclass. Their methods are specific to their own data structure and they are also responsible for bridging the gap to the communication protocols, such as HTTP implemented in RestClientHttp. For example, the OM2M class has the **createApplication()** method where it registers a resource on the broker, or an application by OM2M terminology. The equivalent for FIWARE would be **createEntity()**. Each of them will then use the RestHttpClient class to send the request for resource creation.

The goal is to increase the number of middlewares and protocols, and through these additions the process will be perfected and new use-cases will be considered. At this stage the only implemented middlewares are FIWARE and OM2M, both of them using HTTP. Each protocol will represent a different scenario for utilization.

4.1.2 Metrics

At this stage, we decided to implement the following metrics:

- Publish time
- Subscribe time
- Failed publishes
- Successful publishes
- Total publish time
- Goodput
- CPU usage
- RAM usage

Publish time is defined as the time between publishing the data and receiving the response, as per figure 4.2, whereas subscribe time is measured until the subscriber receives a notification. Goodput is defined as the useful bytes over the time of transmission, with the useful bytes being the payloads of the HTTP packets. These metrics are measured in the metrics class, where the data for their extraction is passed on by the load class. CPU and RAM usage are measured on the machine running the broker through a bash script.

The failed and successful publishes are calculated be each response that is received by the publisher. If a given request is delivered but the response is not received in a given amount of time, that request is considered to have failed. These allow the user to determine reliability of each middleware. A special mention must be made here regarding the time from which a publish is

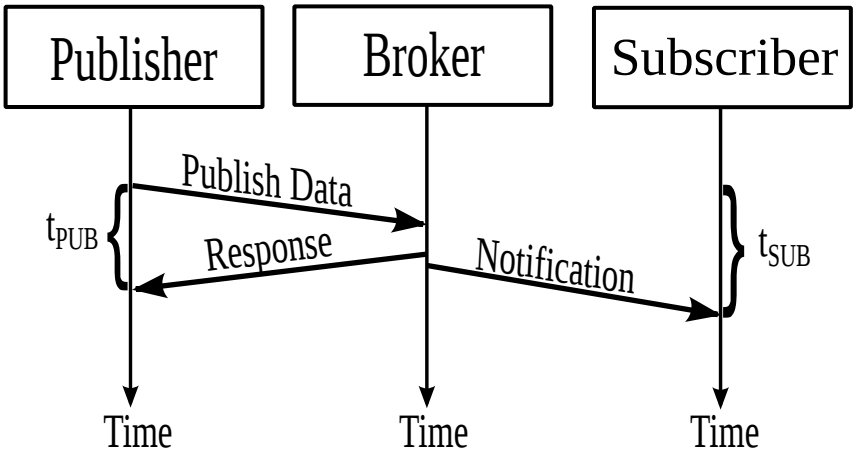

Figure 4.2: Publish and Subscribe times [11]

considered to have failed. This is a parameter that is up to the user to define. A larger timeout may decrease the number of failed publishes, but will also increase the publish time.

Total publish time is the time it takes to publish a certain number of requests, meaning the elapsed time from the sending of the first request to the response of the last request. It can be particularly useful in massive IoT scenarios, where we want to publish mass quantities of information at once, and the time it takes for the entire load is relevant.

These metrics were the first to be implemented, as they seemed the most generic, and are of great relevance to the user, as they allow to determine how much time each request is taking. Also, at the time, we believed that measuring the impact each middleware had on the machine that it was running on was a relevant parameter to access if one could be significantly lighter to use, in case the user faces performances restraints from its machine.

4.1.3 Limitations

In our first approach to implement a solution, we noticed a few shortcomings of our platform. Starting with the structure, it was not obvious what was similar code between middlewares, and this made the job harder for the user to determine exactly what they needed to implement for a new addition. Figure 4.3 shows the differences between the publish methods of FIWARE and PonteHttp. The highlighted section is the section of code that is different and had to be rewritten. Not only was it not clear what could be re-utilized, there was not enough separation between the middlewares and the protocol utilized.

Also, no quantification of request sizes was being made. Different middlewares use different publish structures according to their own specification. Naturally, the larger the payloads the larger the impact on the network and on the space requirements for the broker and the sensors.

No multithreading was implemented. This meant that we could not simulate a scenario with multiple publishes, since in single threaded mode each request will only happen once the previous one has been acknowledged. There should be a way to make parallel requests.

```
public long[] publish(String message, String publishSequenceNumber) {
        long[] returnArray = new long[3];
        Integer payloadSize = 0;
        long elapsedTime = 0;
        String payload = message + "-" + publishSequenceNumber;
        long start = System.currentTimeMillis();
        String[] updateEntityReturn = null;
        updateEntityReturn = updateEntity("entity" + currentEntity.toString()
        payloadSize = updateEntityReturn[0].length();
        String publishResponseStatus = updateEntityReturn[1];
        if (publishResponseStatus.contains("HTTP/1.1 2")) {
                System.out.println("Successful publish ");
                elapsedTime = System.currentTimeMillis() - start;
        }
        else {
                System.out.println("Unsuccessful publish");
                elapsedTime = -1;
        }
        setNextPublishDestinationRoundRobin();
        returnArray[0] = elapsedTime;
        returnArray[1] = payloadSize;
        returnArray[2] = start;
        return returnArray;
}
 Fiware.java     12% L44    Git:threads  (Java/l Server Fly FlyC- Wrap Abbrev)
public long[] publish(String message, String publishSequenceNumber) {
        RestHttpClient client = new RestHttpClient();
        long[] returnArray = new long[3];
        Integer payloadSize = 0;
        long elapsedTime = 0;
        String payload = message + "-" + publishSequenceNumber;
        long start = System.currentTimeMillis();
        payloadSize = payload.length();
        CloseableHttpResponse response = null;
        response = client.put(this.uri + "/resources/" + "resource" + currentR
        String publishResponseStatus = response.getStatusLine().toString();
        if (publishResponseStatus.contains("HTTP/1.1 2")) {
                System.out.println("Successful publish ");
                elapsedTime = System.currentTimeMillis() - start;
        }
        else {
                System.out.println("Unsuccessful publish");
                elapsedTime = -1;
        }
        setNextPublishDestinationRoundRobin();
        returnArray[0] = elapsedTime;
        returnArray[1] = payloadSize;
        returnArray[2] = start;
        return returnArray;
 Ponte.java      33% L39    Git:threads  (Java/l Fly FlyC- Wrap Abbrev)
```

Figure 4.3: Comparison between FIWARE and Ponte publish method

4.2 Second Iteration — Additional Factorization

In the first iteration of the platform, each middleware iteration would be implemented in a single class as per figure 4.1. Each class implemented specific methods regarding the payload to be sent, such as **createApplication**() or **createEntity**(). Since the payload was different between middlewares, it was not possible to create a common method in the Middleware superclass. However, we noticed that the process of making a publish request had a pattern. First, a payload was assembled in accordance to the middleware requirements. This could be a JSON such as the one in listing 4.1.

```
{
        "m2m:cin": {
                "con": "my_message-00000000000",
                "cnf": "application/json",
                "rn": "time_1520895952979"
        }
}
```

Listing 4.1: JSON payload for content instance creation in OM2M

Second, the payload was sent using a certain protocol such as HTTP. Lastly, the response is obtained and added to a list in order to calculate the metrics at the end of the benchmark test. Because of this, we thought it best to re-structure the platform.

4.2.1 Improved Structure

In regards to the publish requests, the main goal in the new structure was to separate the creation of payloads from their sending. In order to accommodate for this, several changes were made. First, the Middleware class was altered so that it now implements the previously described publish structure: assemble the payload, send and wait for a response, pass response to the metrics class. This makes the structure equal among all middlewares. We created seven abstract methods to be implemented by each subclass, as these cannot be factorized due to differing structures and protocols. Second, we further divided the classes so that we could separate what is a global in a middleware, regardless of protocol. This resulted in the class structure visible in figure 4.4.

Taking OM2M as an example, we can see in figure 4.4 three classes pertaining to it: Om2m, Om2mHttp, and Om2mCoap. The OM2M class is responsible for the creation of structures to be sent, such as publish or subscribe structures, regardless of which protocol is used. Each of the other two classes will then implement their own methods regarding the sending of the structures. Since they are the same, this allows the platform to add a new protocol implementation, without having to rewrite the methods for structure creation. However, if the structures to be sent are different between protocols, then they will inevitably need to be re-implemented.

The topmost classes will define the level of abstraction and number of features we can attain. These classes are: Middleware, Load, and Metrics, as these will be used by all middleware implementations. Let's take a closer look at listing 4.2, were we can see the structure of the Middleware class and how it affects the overall platform, and how it limits possible middlewares.

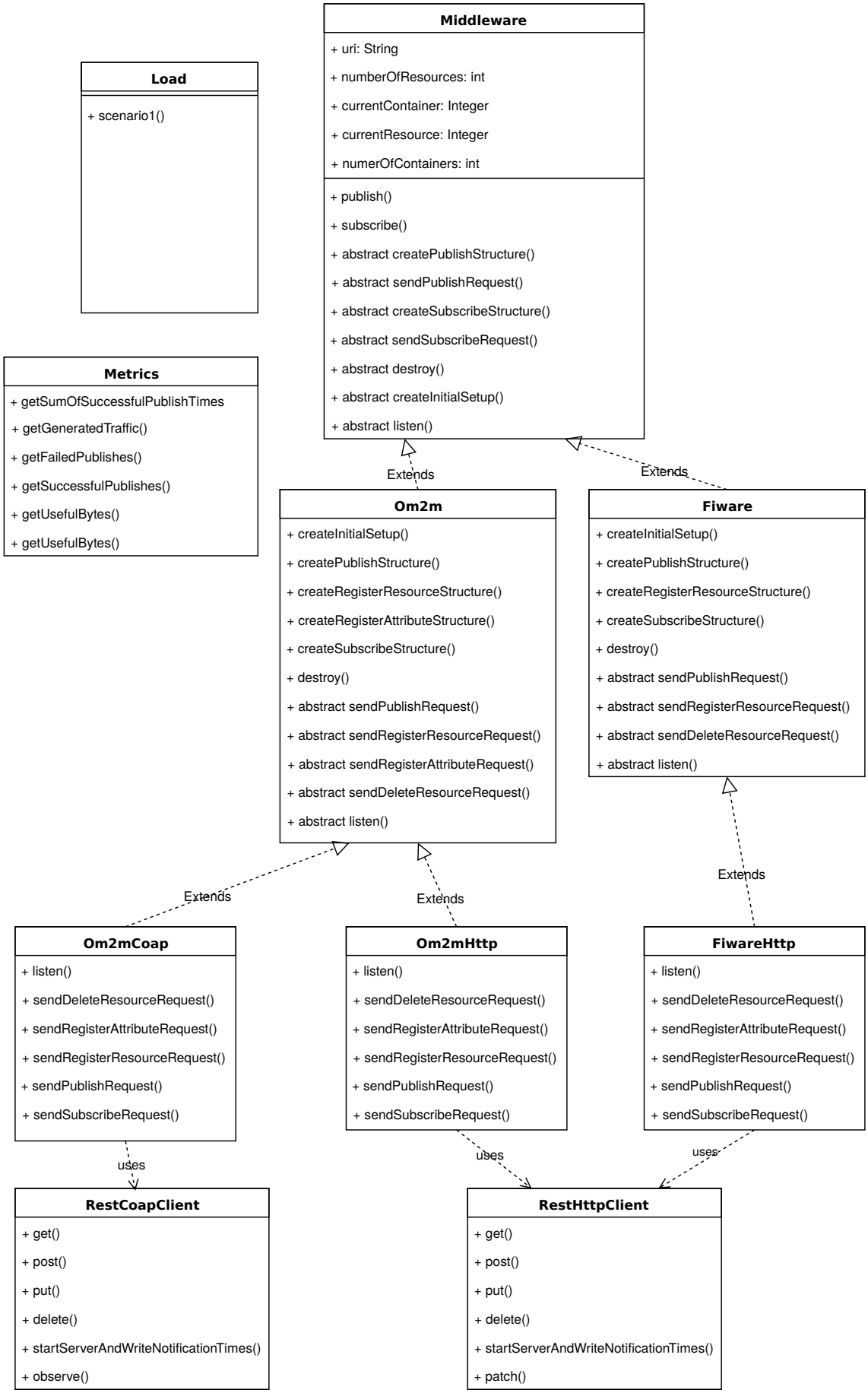

Figure 4.4: Class diagram for the second iteration

```
1  public String[] publish(String message, String publishSequenceNumber) {
2          String[] returnArray = new String[5]; //values for the metrics, will be returned to
               load
3          Integer publishStructureSize = 0;
4          Integer urlSize = 0;
5          long elapsedTime = 0;
6          long start = System.currentTimeMillis();
7          String publishStructure = createPublishStructure("resource" + currentResource.
               toString(), "attribute" + currentAttribute.toString(), message + "-" + "seqNumber
               " + publishSequenceNumber); // creates the structure to be published, such as a
               json or xml
8          publishStructureSize = publishStructure.length();
9
10         String[] publishReturn = sendPublishRequest("resource" + currentResource.toString(),
               "attribute" + currentAttribute.toString(), publishStructure); // sends the
               request
11         String publishStatus = publishReturn[0];
12         urlSize = publishReturn[1].length();
13
14         elapsedTime = System.currentTimeMillis() - start;
15         setNextDestinationRoundRobin();
16         if( publishStatus.contains("Unsucessful")) {
17                 returnArray[0] = publishSequenceNumber;
18                 returnArray[1] = "-1";
19                 returnArray[2] = publishStructureSize.toString();
20                 returnArray[3] = "-1";
21                 returnArray[4] = urlSize.toString();
22         }
23         else {
24                 returnArray[0] = publishSequenceNumber;
25                 returnArray[1] = Long.toString(elapsedTime);
26                 returnArray[2] = publishStructureSize.toString();
27                 returnArray[3] = Long.toString(start);
28                 returnArray[4] = urlSize.toString();
29         }
30         return returnArray;
31 }
```

Listing 4.2: Publish method in the middleware class

First, we have the method **createPublishStructure()**, where we pass the relevant arguments for our structure. These are: resource name, attribute name, message, and sequence number. The sequence number will be concatenated with the message and is only used for matching the publishes with the notifications. The remaining three arguments are the ones we assume should be required most times. We are assuming that middlewares should not require anymore arguments for a publish structure. If a middleware proves this to not be the case, either the method itself will try and generate the missing information, or a rework will have to be done to provide any extra arguments as needed, and existing middlewares will simply not use them. This method will return the structure to be published and can be measured for length. Second, there is the **sendPublishRequest()**, where the previously created structure will be sent using an appropriate protocol. Here we have to account for any protocol that may be used, such as HTTP or CoAP, so the arguments must not be protocol specific. Again, we assume the only arguments necessary

should be the resource and attribute names. At the end we have our return values, which will be passed on to the calling function.

Next we will look at the load class, which will implement any load scenarios we wish to add to our platform. In listing 4.3 a segment pertaining to the load is visible, where we can see how the publishes are sent and what type of sending pattern this represents.

```
1  Thread[] threadArray = new Thread[numberOfThreads];
2  for (Integer i = 0; i < numberOfThreads; i++) {
3          final Integer threadNumber = i;
4          Thread t = new Thread(new Runnable () {
5                  @Override
6          public void run() {
7                  String publishSequenceNumber;
8                  for (Integer currentPublish = 0; currentPublish < publishesPerThread;
                       currentPublish++) {
9                      publishSequenceNumber = String.format("%04d", threadNumber) + String.
                           format("%07d", currentPublish);
10                     //myList.add(myBroker.publish(message, publishSequenceNumber));
11                     try {
12                                     Thread.sleep(minimumWaitTimeBetweenRequests);
13                     } catch (InterruptedException e) {
14                                     e.printStackTrace();
15                     }
16                     myList.add(myBroker.publish(message, publishSequenceNumber));
17                     System.out.println("list size " + myList.size());
18                     System.out.println("length" + publishSequenceNumber.length());
19                   }
20                 }
21         });
22         t.start();
23         threadArray[i] = t;
24 }
```

Listing 4.3: Segment of the **scenario1()** method of the load class

The user will specify the number of requests to send and the number of publishers they wish to simulate. Each publisher will correspond to a thread. The total number of publishes will be divided by the number of threads, and each thread will send its portion of the total load, in sequence, as we can see in figure 4.5. This means that the subscriber will receive several requests from simultaneous publishers, and will use the previously mentioned sequence number to match them. This also means that each publisher will wait until it has received an acknowledgment of its last request until it sends the next one. Each thread will send its requests at a certain variable rate, never exceeding a maximum that is defined by the user. An important detail to notice is that since each request will wait until it has received an acknowledgment, this rate will be capped by the RTT of the connection, and the processing capabilities of the broker.

4.2.2 New Metrics

In addition to the factorization, we also looked to add more metrics to the platform. To maintain modularity, these were added in their own class so that future metrics may be added to the plat-

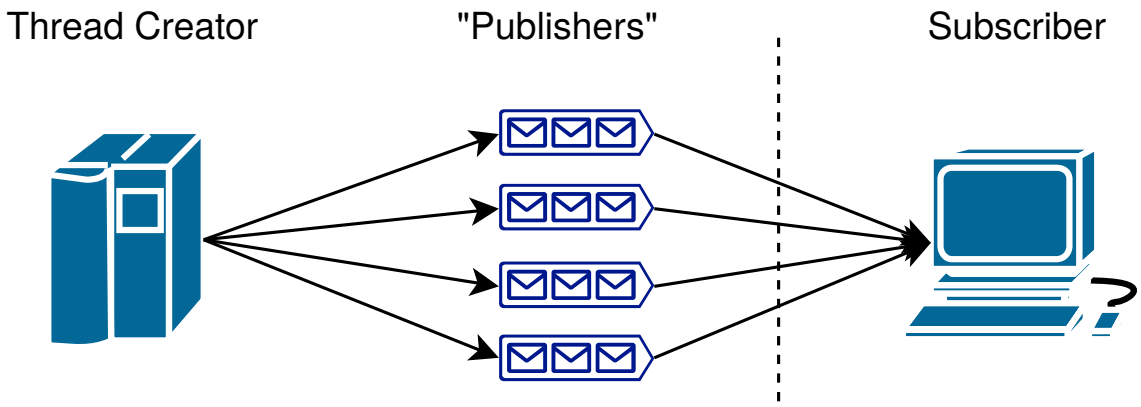

Figure 4.5: Thread model of our workload

form, without interfering with the previous ones. This results in a new block structure visible in figure 4.6. The times are registered in the load class. For each publish in the cycle, the start and

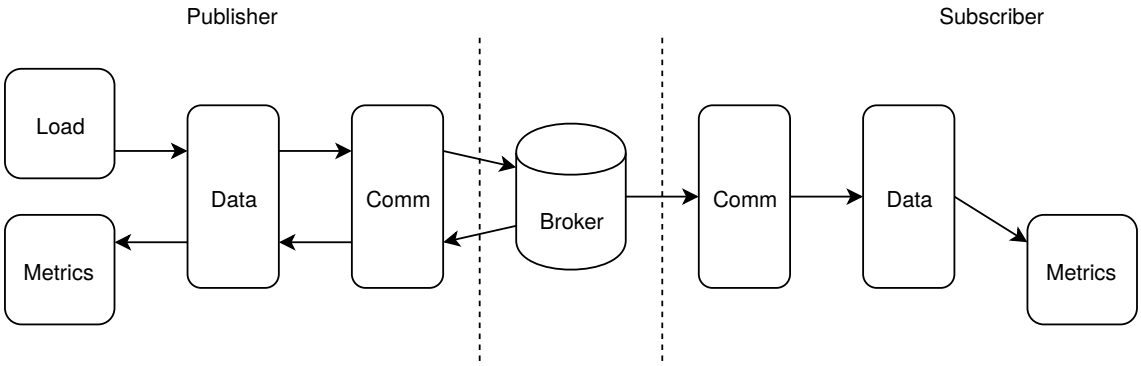

Figure 4.6: Basic block diagram for the second iteration

finish time will be recorded, giving us the publish time. These will be recorded in a list, alongside the structure size, and url size. This list will then be passed on to the Metrics class to calculate all metrics. At this stage, we added:

- Sum of all successful publish times
- Generated traffic
- Size of a publish structure
- Size of the URL
- Size of the actual message

Before proceeding, a brief explanation is required on what each new metric offers in terms of information, and how they are calculated. Starting with the sum of successful publish times, as the name implies, it is the sum of each individual publish time.

Next, we have the generated traffic. It is calculated by dividing each payload size by the time it took to make the publish request. It provides information on how much load is being imposed on the network due to the payloads of each middleware.

The size of the publish structure refers to the size of the structure generated by **createPublishStructure()** method. Each middleware may have its own specification, and each publish request will have to adhere to it. This means for the same message, different middlewares will have requests of different sizes, and will impact the network and sensors differently.

The final two refer only to the size of URL and message. The URL size can be relevant when presented with small messages of size comparable to the URL, as it can have an impact on the overall size of the request.

In addition to the previous metrics, a few python scripts were developed to create boxplots of all the publish and subscription times. These were done in python as opposed to java due to the fact of being considerably easier and less time-consuming. A few examples can be seen in the results section.

4.2.3 Obstacles that lead to framework changes

4.2.3.1 OM2M HTTP and CoAP

In order to measure the subscribe time of the publish requests, the time difference between when the publish was sent by the publisher, and when the notification was received by the subscriber, must be measured. This requires the subscriber to register each time a notification arrives, so that it may be compared with the original sent time. Our new structure predicts this, as the middleware class implements a generic **subscribe()** method. Similarly to the **publish()** method, this method's original goal was to provide the structure for a generic subscription request. Typically this means the creation of a subscription structure and its sending. Following this, a server would be created to listen and register all incoming requests and register its times.

When we proceeded to add Om2mHttp and Om2mCoap, a problem arose. In order for a subscription to be created, the broker must first contact the subscriber in question in the designated address to verify that it is in fact listening to requests. This means that the listener had to be created before the subscription was made. To better accommodate this types of cases, we thought best to separate the subscription from the listener, so that the listener may always be started first, even if it is not required.

4.2.3.2 OM2M MQTT

When adding support for MQTT with OM2M a few challenges arose. OM2M does not include a MQTT broker, and it requires a separate one in order to function. For the purpose of this thesis, mosquitto[1] was used. In figure 4.7 we can see a diagram for the message flow that arises with this type of setup.

[1] https://mosquitto.org/

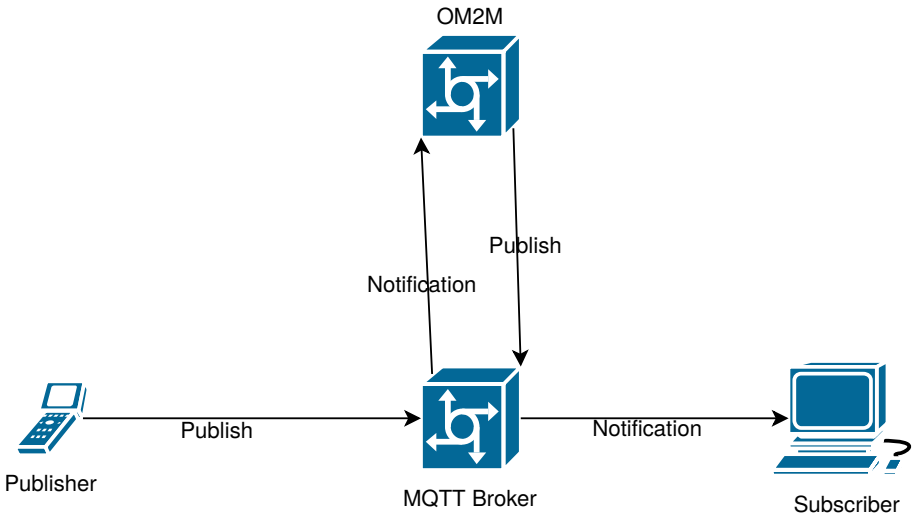

Figure 4.7: Diagram for OM2M MQTT message flow

The messages from the publisher will go directly to the MQTT broker, on a topic of the form `/oneM2M/req/<originator>/<target-id>/<serialization-format>`, where the originator and target-id refer to the identifications tags for the origin of the request, in this case the publisher, and the destination. The serialization-format can have one of two values: json, or xml. OM2M will be subscribed to this topic and will, therefore, receive a notification. In order to notify any subscribers, it will publish on a topic of the form `/oneM2M/resp/<target-id>/<originator>/<serialization-format>`, were the fields have the same meaning as the previous topic, but the originator and target-id tags are reversed. This means that the OM2M broker will be acting as both a publisher and a subscriber.

As it has been mentioned, one of the goals of our framework is to provide code re-usability. To provide code re-usability, we aimed for our classes to be modular, and to follow the architecture presented in figure 4.6. In theory, to add support to a new protocol for an existing middleware, all that should be required is to create the class that implements the link between the data and communication blocks. Looking back on figure 4.4, this would mean creating a Om2mMqtt class which extends the OM2M class and uses a generic MQTT client. For this to be possible, we need the structures created by the OM2M class to be independant of the protocol. However, this is not the case with OM2M. Even though the structures are the same between HTTP and CoAP, with MQTT they are different. In listing 4.4 we can see an example of a payload structure for OM2M with HTTP, and in listing 4.5 for MQTT. These are significantly different. Therefore, the existing method that creates a publish structure cannot be reused, and must be re-implemented. Those methods are: **createPublishStructure**(), **createSubscribeStructure**(), **createRegisterAttributeStructure**, and **createRegisterResourceStructure**().

```
{
        "m2m:cin": {
                "con": "RFYGrtmPTX-seqNumber00000000000",
                "cnf": "application/json",
                "rn": "time_1528643928716"
        }
}
```

Listing 4.4: JSON structure for OM2M HTTP publish

```
{
        "m2m:rqp": {
                "m2m:op": 1,
                "m2m:ty": 4,
                "m2m:pc": {
                        "m2m:cin": {
                                "con": "ZWDpOJCWgu-seqNumber00000000000"
                        }
                },
                "m2m:fr": "admin:admin",
                "m2m:rqi": 123456,
                "m2m:to": "/in-cse/in-name/resource0/attribute0"
        }
}
```

Listing 4.5: JSON structure for OM2M MQTT publish

Another difference that came up when implementing MQTT support for OM2M was in regards to the **sendDeleteResourceRequest()** method. Previously, with CoAP and HTTP this method had no arguments as there was no payload to be sent. The information on which resource was to be deleted is present in the URL, and using the delete option of both protocols, no more information is necessary. However, MQTT has no such option, and OM2M uses an external broker to process MQTT request. Therefore, the information must be passed on by some payload, so a change was made to the method to accommodate this.

4.2.4 Limitations

One of the consequences of adding multithreaded support is that it is no longer guaranteed that the notifications will arrive in the same order by which the publishes were sent. Therefore, we need a way to map each notification to its respective publish. We did this by adding a sequence number to each publish request, that identifies the publisher and the order. For this to work, a few assumptions are made in regards to the notifications. First, we assume the sequence number will always be present in the notification. This is to be expected, as a subscriber should want to know the changes that were made to the resource that it subscribed to. If we consider a situation where the subscriber is notified of a change, but does not inform on what that change was, our method will not work and the subscribe time will not be possible to calculate. Second, it implies that messages will always have a minimum length. These extra bytes should be taken into account, especially when the messages that we want to send are of comparable length to the sequence number.

In terms of packet analysis, it is currently being done after the structure is assembled and before being sent. This means that it is being performed above the application layer, and we are limited to the information that we are able to extract from that level. Therefore, we can extract information regarding the structures each middleware assembles, but nothing on how an application level protocol, such as HTTP, is assembling its request. Meaning that information on the total packet size and additional headers is lost. This extends to any lower level such as the transport of network layer. This is due to the fact that the platform was created to be generic, and individual packet analysis would require specific treatment regarding each middleware and each protocol. This has a direct impact in the generated traffic metrics. Providing some sort of lower-level analysis will allow for a much more intuitive metric, and more representative of the actual consumed bandwidth.

Chapter 5

Results

In this section we will provide the results that were obtained across multiple experiments with differing conditions such as message size and rate. It should be noted that the primary goal of the obtained data is not to determine which middlewares are better in which conditions. Rather, we want to use them to validate the platform as an effective tool in order to perform benchmarks, and show what types of comparisons can be made across the different middlewares. A small analysis can be made on the results, but this will be a secondary goal.

We will divide this chapter into multiple sections, each pertaining to the main variable that was changed, such as thread number or message size.

All of the experiments were conducted on the same machine. This machine acted simultaneously as the publisher, broker, and subscriber. While this may not be a real-world setup, it is easier to run several different experiments which better showcase the advantages of our platform, such as what type of information can be extracted from the experiments and the ease of running several different experiments by changing a few parameters. Since the main goal is to highlight our platforms capabilities, we feel this is a reasonable compromise.

5.1 Setup

The machine is a laptop with an Intel Core i5-7200 CPU clocked at 2.5 GHz, dual-core with hyperthreading, for a total of four threads. It has 8GB of RAM. The operating system used for the FIWARE experiments was CentOS 6.9 and the remainder on Ubuntu 16.04 LTS 64-bit. This was because FIWARE does not have an implementation for Ubuntu, requiring it to be built from the source-code. We felt this would take too much time, and that a different operating system for one of the middlewares would not detract from our conclusions.

The resulting platform proved to be very efficient in making experiments and obtaining results. We were able to perform several experiments by merely changing a few parameters. In terms of configuration, the user has to input the following:

- Which middleware will be used
- Broker IP address and port

- Subscriber IP address and port

Then, in order to define our load, the user has to choose:

- Number of resources
- Number of attributes
- Number of publishes
- Number of threads
- Maximum message rate
- Message length

Both the configuration and load parameter are all defined by variables in the main class, so that they can be easily changed. When the platform reaches a more finalized state, this can be moved to a configuration file for easier access. Assuming all brokers are running and have been properly configured, once the user has defined the load parameters, they only need to switch the middleware to be used, and its location as well as the subscriber.

After the experiments are run, the user gets a few files which contain all the information gathered. In these files we have all the publish and subscribe times, including which requests were unsuccessful. From here, a few python scripts generated the boxplots that we will see further on. Python was used due to being easier and less time-consuming than java. In order to automate the process, the results have to be placed in a certain folder hierarchy, so that all the graphs can be generated automatically.

5.2 Middleware comparison

The first experiment conducted was with 1000 requests, each request carrying a message of 22 bytes, including the tag for the sequence number, at a maximum message rate of 100 messages/s. All publish requests were successful. We are comparing three middlewares with the HTTP protocol. We chose HTTP as it is present in three middlewares, and allows us to make a more meaningful comparison. In figure 5.1 we see the publish times pertaining to each configuration. Immediately, we can see that most OM2M requests took longer, with relatively high variability between about 8 and 18 milliseconds, when compared to FIWARE, which presents itself with almost all requests slightly above 5 milliseconds. Ponte presented a higher variability than FIWARE, between 5 and 10 milliseconds, but still lower than OM2M.

Following this, we have the subscribe times in figure 5.2. The same story repeats itself when comparing FIWARE and OM2M. It should be noted that PonteHttp has no subscribe times as the way for the subscriber to know the state of a given resource is through GET requests, which depends on the polling rate and falls outside the scope of a publish/subscribe model.

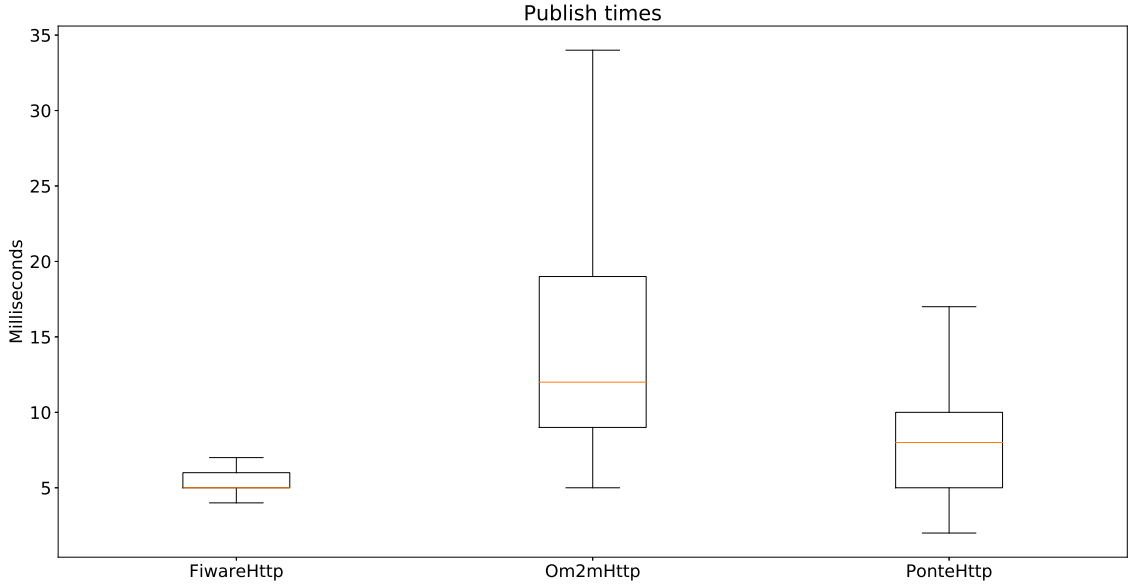

Figure 5.1: Publish times for one thread with 22 byte messages

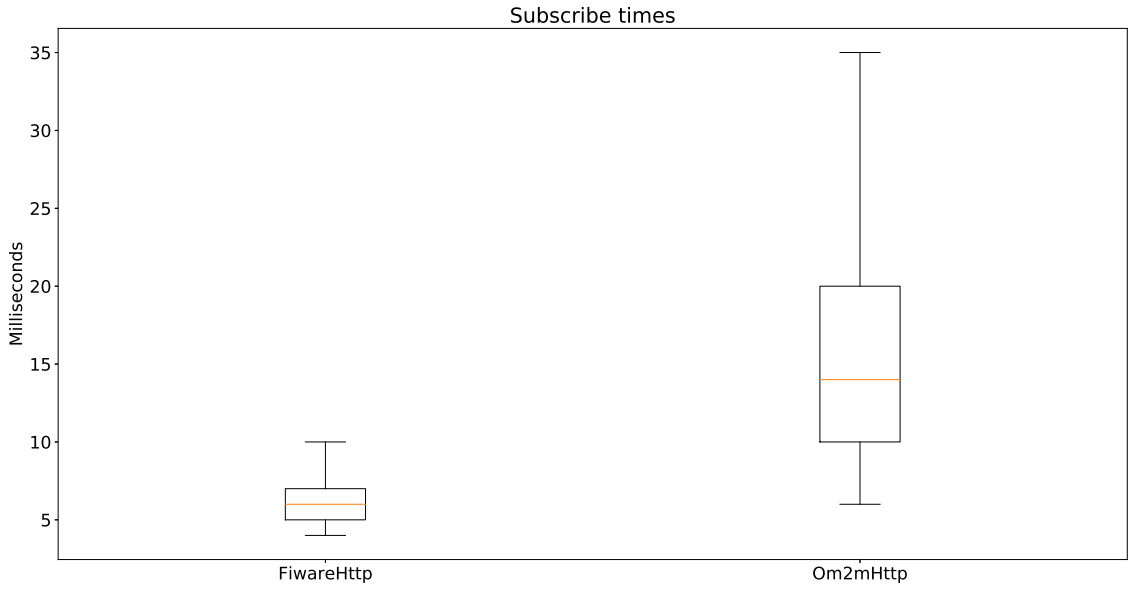

Figure 5.2: Subscribe times for one thread with 22 byte messages

In table 5.1 we have additional metrics provided to us by the metrics class, which we considered more relevant to point out. The sum of successful publish times tells us what we were expecting from the previously seen boxplots, where the total was higher for OM2M. The same is true for the total publish time. We should note here a discrepancy between these two metrics, as the total time is larger by about 10 seconds when compared to the sum. This is due to the message rate. By limiting the platform to a given message rate, each request will add a certain amount of idling time. At a message rate of 100 per second, this means each thread will wait for 10 ms between requests. Since there are 1000 requests, $10ms * 1000 = 10s$, which accounts for the extra time. This is a limitation, as the message rate is not taking into account the time to make the request, and is something to be addressed in future work.

In terms of sizes, the OM2M presents the largest structure, with Ponte being the smallest one. This leads us into the last point, the generated traffic. Since this is calculated with the sizes of the publish requests divided by the publish times, Ponte presented the lowest traffic, due to its smaller footprint in terms of structure size.

Metrics	FIWARE	OM2M	Ponte
Sum of successful publish times (ms)	7681	15510	8276
Total publish time (ms)	17869	26157	18683
Size of a publish structure (Bytes)	74	104	31
Generated Traffic (KB/s)	9.63	6.71	3.74

Table 5.1: Results with 22 byte messages for one thread with HTTP

5.3 MQTT comparison

We decided to make an extra experiment comparing MQTT to utilize RabbitMQ, and better showcase the versatility of our platform. This experiment consisted of 1000 publishes, 22 byte messages, at 100 messages per second using just one thread. All publishes were successful. In figure 5.3 we can see the publish times for the three middlewares. OM2M has a higher variability of publish times, with some exceeding 20ms, being the ones that took longest. However, it is not far off from RabbitMQ. For the subscribe times in figure 5.4, both Ponte and RabbitMQ took low amounts of time, with OM2M having nearly half of its requests between 20 and 40 ms, considerably higher than the other two.

Metrics	OM2M	Ponte	RabbitMQ
Sum of successful publish times (ms)	19053	8376	17934
Total publish time (ms)	29670	18698	28267
Size of a publish structure (Bytes)	186	31	31
Generated Traffic (KB/s)	9.76	3.70	1.72

Table 5.2: Results with 22 byte messages for one thread with MQTT

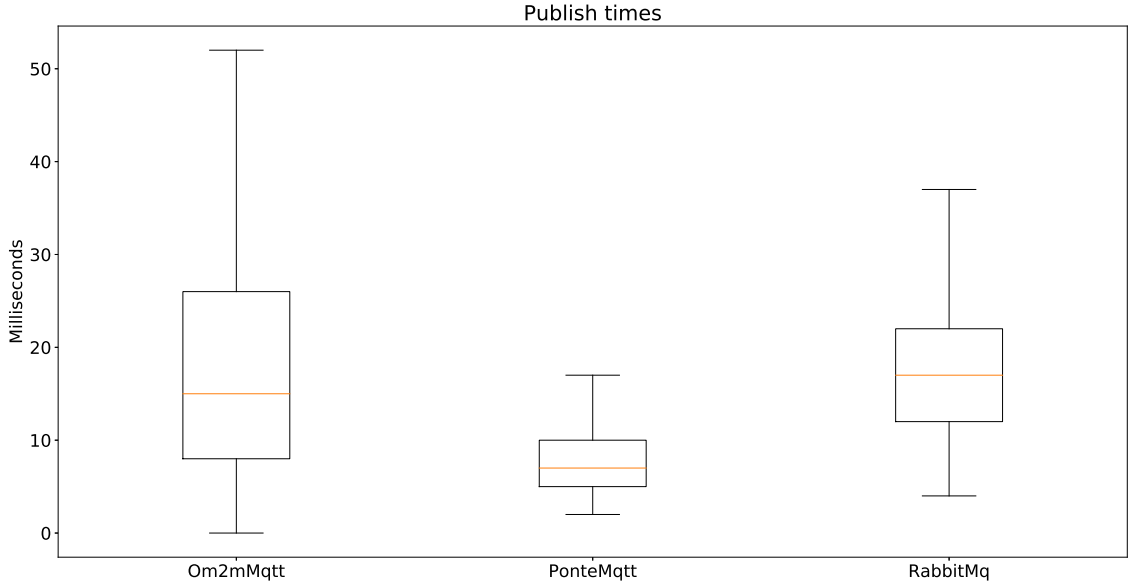

Figure 5.3: Publish times for different midllewares using MQTT

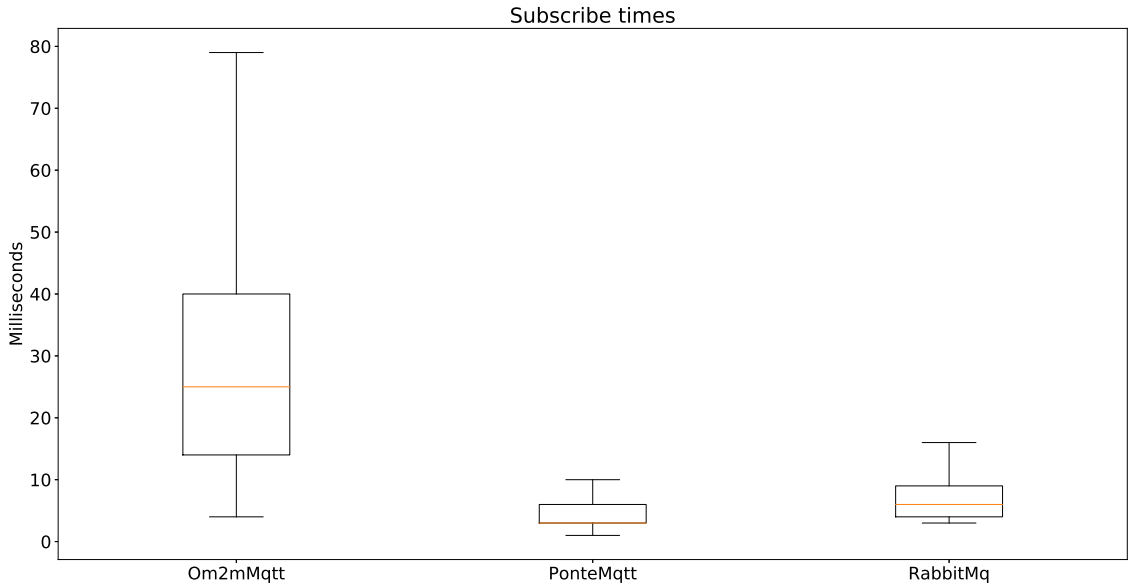

Figure 5.4: Subscribe times for different midllewares using MQTT

In table 5.2 we can see the remaining metrics. The most noteworthy being the publish structure size. OM2M stands out as having a significantly larger structure size of 186. Of course, this has an impact on the generated traffic, which is higher for OM2M as expected.

5.4 Communication protocol comparison

In this section we made a comparison between the different protocols, using the same middleware. OM2M was selected because it supports all three, enabling a more interesting comparison. As before, the experiment was comprised of 1000 requests, each with 22 bytes, using one thread. The maximum message rate was 100 messages/s. In figure 5.5 we can see a comparison between the publish times for the three implementations. CoAP presents the lowest times, with HTTP being just slightly higher overall. MQTT had the highest times and variability. This could be because

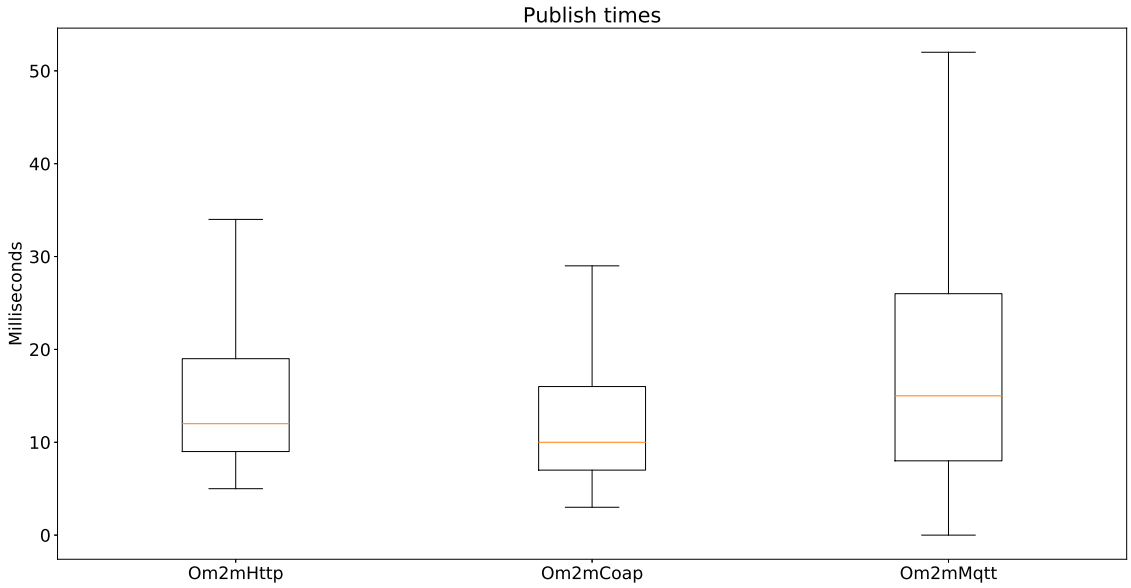

Figure 5.5: Publish times for OM2M with 22 byte messages and one thread

OM2M does not implement MQTT by itself, requiring an external broker to handle the requests. Having two brokers can have an impact on the publish times. For the susbcribe times in figure 5.6, the results are similar.

In table 5.3, we can see the remaining metrics. The most noteworthy is the publish structure size for MQTT, which is not the same as HTTP and CoAP. Since its publish times are also larger, it lowers the generated traffic.

5.5 Security: HTTP vs HTTPS

In terms of benchmarking security, we used FIWARE to make a comparison between HTTP and HTTPS as it is the only one we were able to configure to use HTTPS. It should be noted, however,

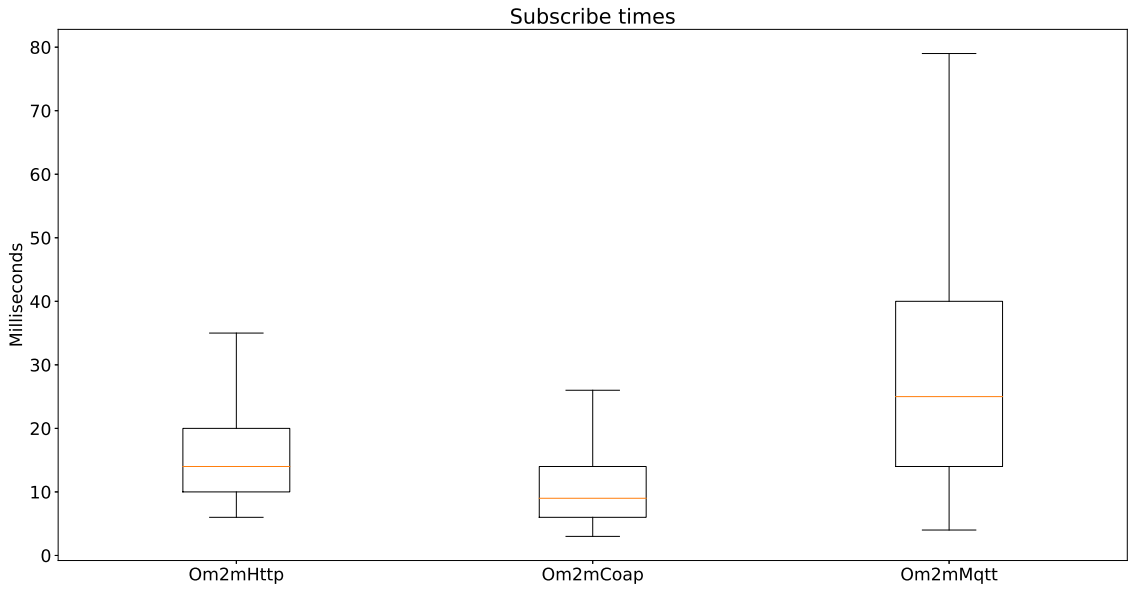

Figure 5.6: Subscribe times for OM2M with 22 byte messages and one thread

Metrics	HTTP	CoAP	MQTT
Sum of successful publish times (ms)	15510	12574	19053
Total publish time (ms)	26157	22984	29670
Size of a publish structure (Bytes)	104	104	186
Generated Traffic (KB/s)	6.70	8.27	9.76

Table 5.3: Results for OM2M with 22 byte messages for one thread

that OM2M also supports it. As before, 1000 publish requests were made, each with 22 byte messages. Only one thread was used. All publishes were successful.

Through analysis of figure 5.7, we can see an increase in total publish time. This is expected as HTTPS takes much longer to establish connections due to the exchange in certificates. The same is true for the subscribe times, in figure 5.8.

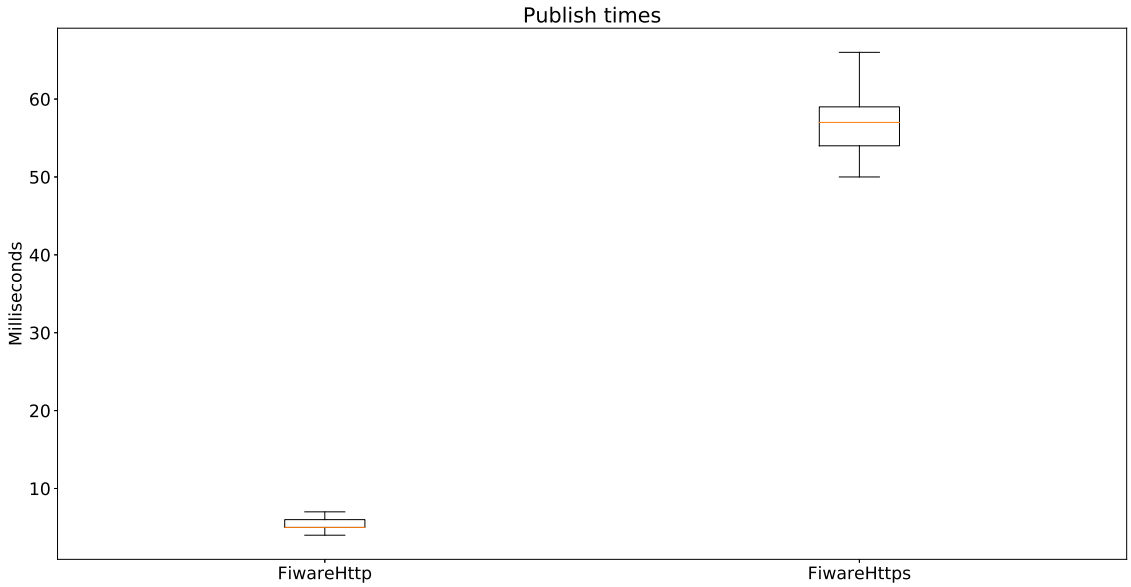

Figure 5.7: Publish times for FIWARE with HTTP and HTTPS

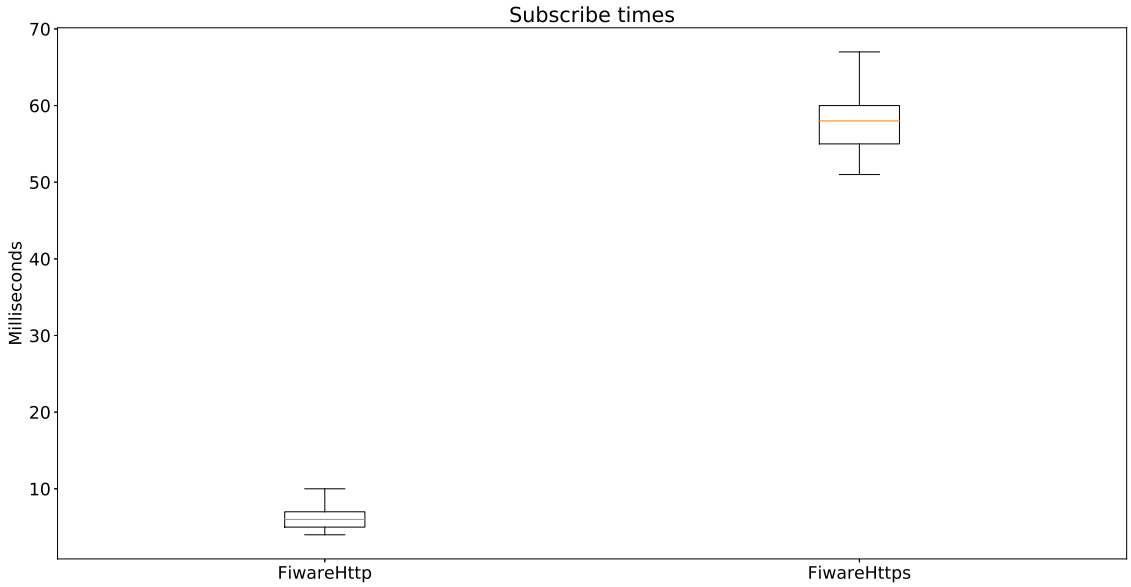

Figure 5.8: Subscribe times for FIWARE with HTTP and HTTPS

Table 5.4 we can see the remaining metrics for our experiment. The generated traffic is decreased, as each publish is the same size but taking a larger amount of time. However, this can

be misleading, as the traffic is only measured on the application layer, meaning any additional overhead caused by the SSL/TLS encryption will not be considered. This is a shortcoming of the current implementation for generated traffic, and one that must be addressed in future work. All the publishes were successful, and all sizes were equal, due to being the same middleware and therefore the same structures and URL. Again, publish structure only takes into account the specific middleware structure, and not any overhead caused by the protocol.

Metrics	HTTP	HTTPS
Sum of successful publish times (ms)	7681	59269
Total publish time (ms)	17869	69497
Size of a publish structure (Bytes)	74	74
Generated Traffic (KB/s)	9.63	1.24

Table 5.4: Results with 22 byte messages for FIWARE with HTTP and HTTPS

5.6 Message size

Here, we kept a similar setup, but started varying the message size. OM2M with HTTP was used as before. This time, the parameters were 1000 requests, 1 thread, but no maximum message rate. The reason for this is we wanted to see if the increased sizes would have an appreciable effect on the total publish times, without any hindrance from the message rate. All publishes were successful.

Both the publish times in figure 5.9 and subscribe times in figure 5.10 depict the same pattern. The times are very constant, with a slight increase towards the end. The most noticeable difference is when the message size grows to 100000 bytes, which is an extraordinarily large payload.

We are omitting here the remaining metrics as they did not provide any noteworthy results.

5.7 Multiple publisher thread comparison

We proceeded to conduct the same experiment for OM2M with HTTP, but this time with a variable thread number. This means we again have 1000 requests, 22 byte messages, the same message rate of 100 messages/s, and variable publisher thread count. In figure 5.11 we see such a graph for Om2mHttp publish times with one, two, five, and ten threads and in figure 5.12 we can see the subscribe times. In both figures we see a natural increase in times, as each broker now has to handle multiple requests in parallel, which results in longer times.

In table 5.5, we can see the metrics for the OM2M results. Starting from the top, the sum of successful publish times increased with the number of threads as expected. The total publish time decreased significantly until five threads, with a small benefit with ten threads. This being a four thread CPU, this is to be expected as there should be little to no benefit from over four threads in parallel.

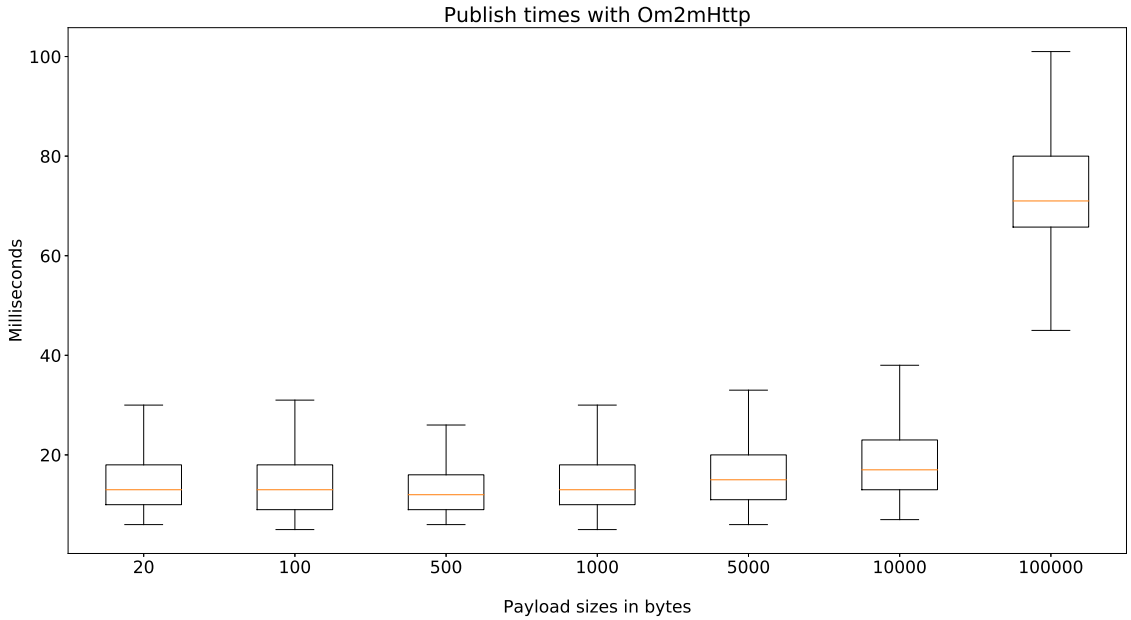

Figure 5.9: Publish times for OM2M with HTTP with variable size

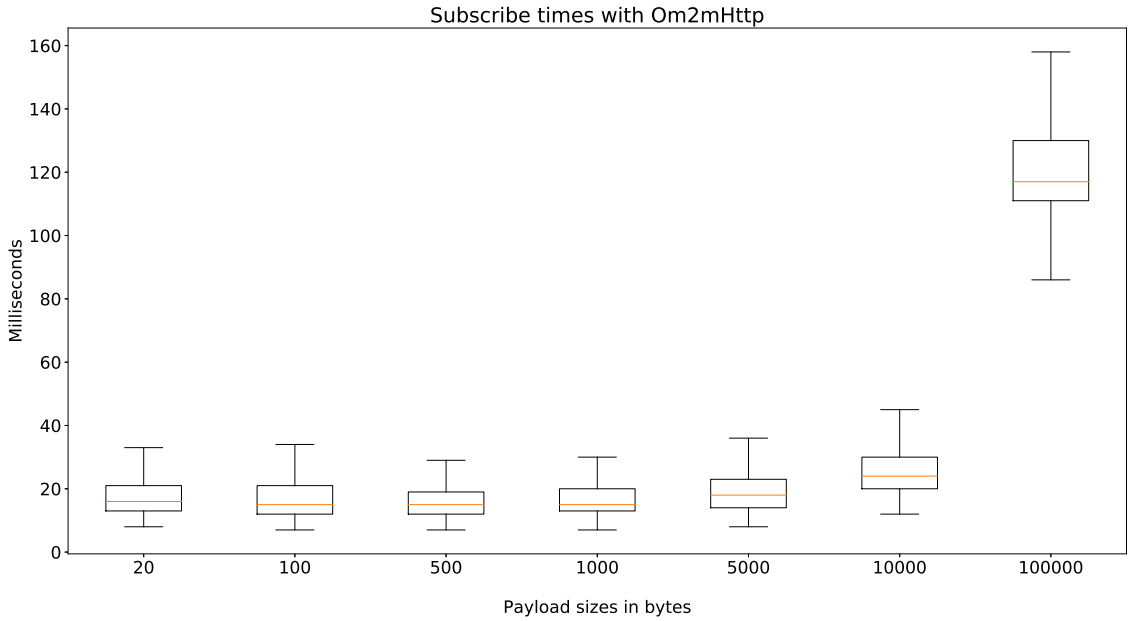

Figure 5.10: Subscribe times for OM2M with HTTP with variable size

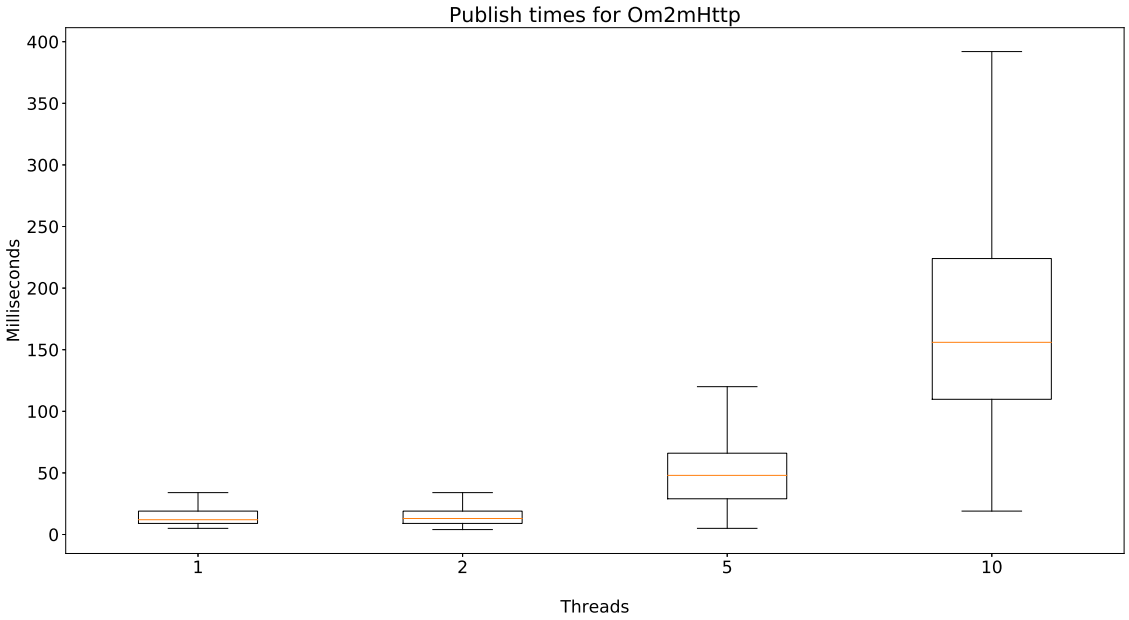

Figure 5.11: Publish times for OM2M with HTTP with multiple thread numbers

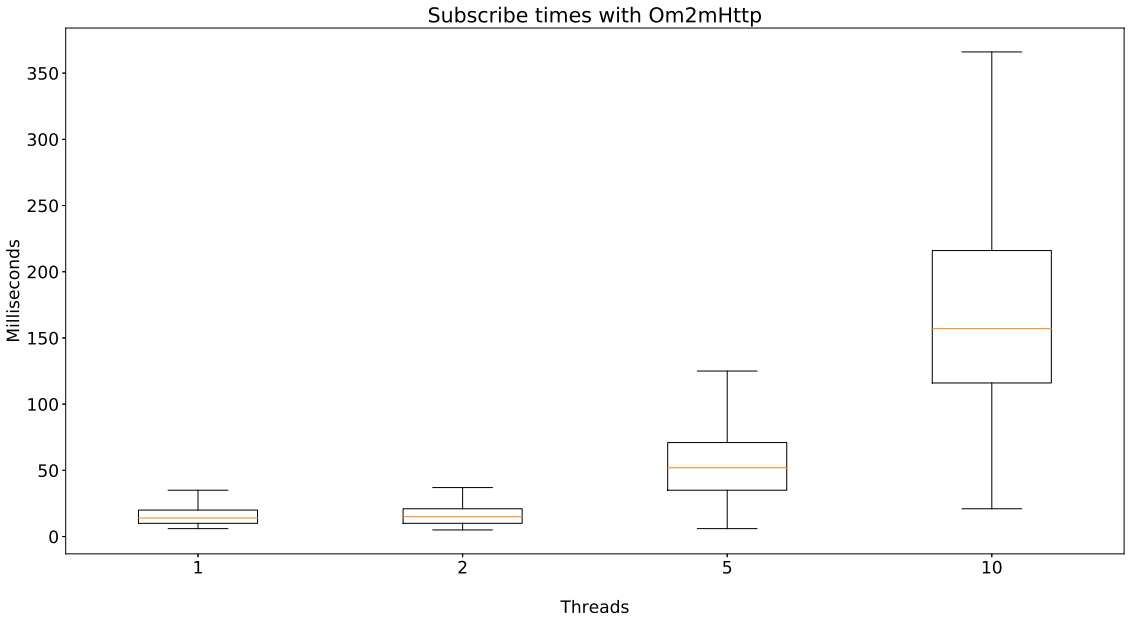

Figure 5.12: Subscribe times for OM2M with HTTP with multiple thread numbers

	Threads			
Metrics	**1**	**2**	**5**	**10**
Sum of successful publish times (ms)	15510	15151	48141	2832748
Generated Traffic (KB/s)	6.70	6.78	2.02	0.03
Failed publishes	0	12	63	96
Successful publishes	1000	988	937	904
Total publish time (ms)	26157	13319	12947	436212

Table 5.5: Results with 22 byte messages for Om2mHttp across multiple thread numbers

When it comes to the number of failed and successful publishes, we see a trend. As we increase the thread number, the number of failed publishes increases. This suggests the broker presents some inability to handle a large number of parallel requests. This indicates a reliability issue for OM2M in case of large parallel loads.

5.8 Multiple subscriber thread comparison

Here we repeated the same setup as the previous experiment. Using again OM2M with HTTP, making 1000 requests, 22 byte messages, the same message rate of 100 messages/s, and variable subscriber threads. Both figures 5.13 and 5.14 show fairly uninteresting results, with a very low variability between thread times. Nonetheless, we decided to include this experiment to showcase how we simulate several subscribers. We limited the experiment to 20 threads as the machine were we conducted the experiments was struggling with a higher thread count.

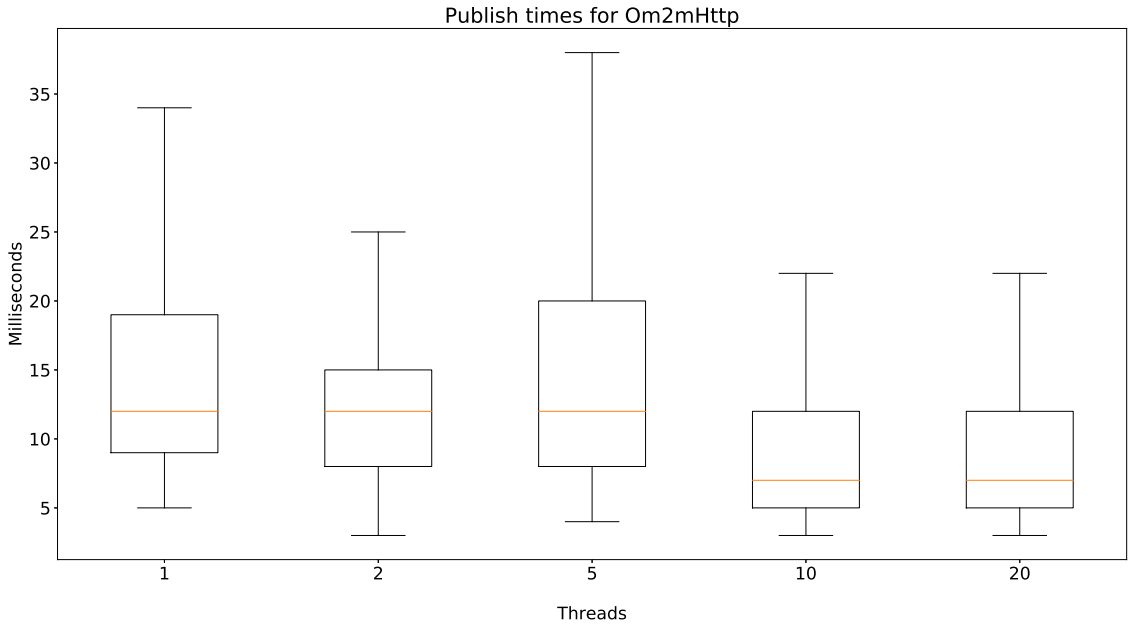

Figure 5.13: Publish times for OM2M with HTTP with multiple subscriber thread numbers

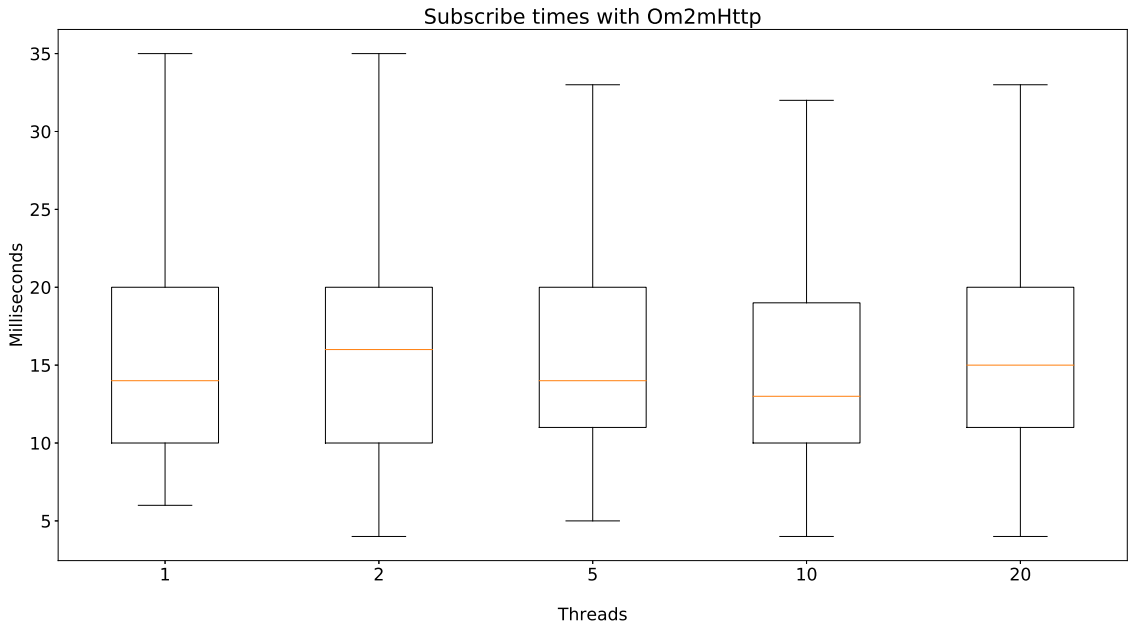

Figure 5.14: Subscribe times for OM2M with HTTP with multiple subscriber thread numbers

Chapter 6

Conclusions

By studying the different approaches to the benchmarking process, we have learned that this is not a trivial process and several factors must be taken into account to ensure we have the conditions necessary for a fair comparison. We have also seen several attempts to benchmark different IoT solutions and observed that they may not be comparable due to differing methodologies. Starting from previous experiments, we identified the common aspects of the benchmarking process and factorized them. From this factorization stemmed a framework that streamlines the entire benchmarking process, due to being faster and easier to add additional middlewares. We conducted several experiments to validate our platform, and show what kind of information we can extract from it, as well as its usability. We obtained a good level of results, which will allow for future users to more easily test different middlewares, as well as providing some common ground so that different experiments may be more directly compared.

However, the process has its trade-offs. Since we are trying to make the platform as generic as possible, it is very hard, if not impossible, to include specific metrics. Therefore, some information will be lost regarding specific middleware details.

For future work, we want to improve on the metrics that are currently implemented. A lower-level analysis on the TCP level would be a great improvement. Of course, this is difficult due to the generic nature of our framework, but some work must be made here as it is an important aspect in order to quantify bandwidth consumption from each implementation. This would also be a good improvement for the existing generated traffic metric, as its current state does not provide intuitive information, and requires some improvement. Also, we would like to extend the usability of this platform from simulated setups to real-world IoT systems. Currently, the number of publishers and subscribers are simulated with threads representing a single publisher of subscriber. While some relevant information can be extracted from such a setup, it would be very beneficial to be able to benchmark existing systems, so that we can have a much more accurate idea of how a system is performing. This will imply having each publisher and subscriber have its own load and being managed by a central machine, so that in the end we are able to measure and aggregate all of the generated data, since each publisher will be on a separate machine. We believe that it would be the main structural change in a third, or possibly fourth iteration of our solution.

Bibliography

[1] M. Dixit, J. Kumar, and R. Kumar, "Internet of things and its challenges," in *2015 International Conference on Green Computing and Internet of Things (ICGCIoT)*, pp. 810–814, Oct. 2015.

[2] P. Suresh, J. V. Daniel, V. Parthasarathy, and R. H. Aswathy, "A state of the art review on the Internet of Things (IoT) history, technology and fields of deployment," in *2014 International Conference on Science Engineering and Management Research (ICSEMR)*, pp. 1–8, Nov. 2014.

[3] N. Fantana, T. Riedel, J. Schlick, S. Ferber, J. Hupp, S. Miles, F. Michahelles, and S. Svensson, "Internet of Things - Converging Technologies for Smart Environments and Integrated Ecosystems," pp. 153–204, Jan. 2013.

[4] M. A. Razzaque, M. Milojevic-Jevric, A. Palade, and S. Clarke, "Middleware for Internet of Things: A Survey," *IEEE Internet of Things Journal*, vol. 3, pp. 70–95, Feb. 2016.

[5] L. Zilhão, R. Morla, and A. Aguiar, "A modular tool for benchmarking iot publish-subscribe middleware," *2018 IEEE 19th International Symposium on A World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, 2018.

[6] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The Many Faces of Publish/Subscribe," *ACM Comput. Surv.*, vol. 35, pp. 114–131, June 2003.

[7] P. A. Bernstein, "Middleware: A Model for Distributed System Services," *Commun. ACM*, vol. 39, pp. 86–98, Feb. 1996.

[8] G. Fersi, "Middleware for Internet of Things: A Study," in *2015 International Conference on Distributed Computing in Sensor Systems*, pp. 230–235, June 2015.

[9] J. Hennesey and D. Patterson, *Computer Architectures - A Quantitative Approach*. Morgan Kaufmann, 2011.

[10] A. Sangroya and S. Bouchenak, "A Reusable Architecture for Dependability and Performance Benchmarking of Cloud Services," in *Service-Oriented Computing – ICSOC 2015 Workshops*, Lecture Notes in Computer Science, pp. 207–218, Springer, Berlin, Heidelberg, Nov. 2015.

[11] J. Cardoso, C. Pereira, A. Aguiar, and R. Morla, "Benchmarking IoT middleware platforms," in *2017 IEEE 18th International Symposium on A World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, pp. 1–7, June 2017.

[12] C. Pereira, J. Cardoso, A. Aguiar, and R. Morla, "Benchmarking Pub/Sub IoT middleware platforms for smart services," *Journal of Reliable Intelligent Environments*, pp. 1–13, Feb. 2018.

[13] A. Shukla, S. Chaturvedi, and Y. Simmhan, "RIoTBench: A Real-time IoT Benchmark for Distributed Stream Processing Platforms," Jan. 2017.

[14] D. Happ, N. Karowski, T. Menzel, V. Handziski, and A. Wolisz, "Meeting IoT platform requirements with open pub/sub solutions," *Annals of Telecommunications*, vol. 72, pp. 41–52, Feb. 2017.

[15] R. T. Fielding, J. Gettys, J. C. Mogul, H. F. Nielsen, L. Masinter, P. J. Leach, and T. Berners-Lee, "Hypertext transfer protocol – http/1.1," RFC 2616, RFC Editor, June 1999. http://www.rfc-editor.org/rfc/rfc2616.txt.

[16] Z. Shelby, K. Hartke, and C. Bormann, "The constrained application protocol (coap)," RFC 7252, RFC Editor, June 2014. http://www.rfc-editor.org/rfc/rfc7252.txt.

[17] J. Guth, U. Breitenbücher, M. Falkenthal, F. Leymann, and L. Reinfurt, "Comparison of IoT platform architectures: A field study based on a reference architecture," in *2016 Cloudification of the Internet of Things (CIoT)*, pp. 1–6, Nov. 2016.

[18] C. W. Wu, F. J. Lin, C. H. Wang, and N. Chang, "OneM2M-based IoT protocol integration," in *2017 IEEE Conference on Standards for Communications and Networking (CSCN)*, pp. 252–257, Sept. 2017.

[19] K. Vandikas and V. Tsiatsis, "Performance Evaluation of an IoT Platform," in *2014 Eighth International Conference on Next Generation Mobile Apps, Services and Technologies*, pp. 141–146, Sept. 2014.

[20] M. Collina, "Github - eclipse/ponte: Ponte project." https://github.com/eclipse/ponte. (Accessed on 05/27/2018).